КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Информатика

Программирование на языке C++

составитель Гайнутдинова А.Ф.

Казанский федеральный университет

2010

Оглавление

Глава 1

Основные конструкции языка C++

1.1 Программирование

Цель программирования – облегчение жизни людей путем автоматизации рутинных, стандартизованных форм их интеллектуальной деятельности. Происходит частичная замена человека автоматом в специфической, уникальной сфере деятельности – мышлении. При этом освобождается творческая активность человека для решения действительно сложных проблем.

Программирование предполагает создание искусственных объектов – абстрактных моделей, имитирующих строение и поведение реальных, и оперирование такими моделями. Моделирование – задача любой науки. Но для программиста содержательная сложность предмета описания усугубляется тем, что результатом его работы должна быть компьютерная программа, ориентированная на исполнение машиной. Поэтому описание моделей из произвольных сфер деятельности должно производиться не только на предельно формальном, но и крайне бедном языке.

Математический язык, используемый для описания реальной области деятельности, называют *языком спецификации* (точного определения), а саму эту область – *предметной*. Создание сложных, реально работающих программ невозможно без постепенного уточнения описания модели, начиная со спецификаций, ориентированных на использование человеком. Имеет место следующая схема:

1. “язык реальности” – все многообразие, воспринимаемое органами чувств;
2. естественные языки – устная, письменная речь (искусство);
3. формально-математические языки спецификаций (наука);

4. языки программирования высокого уровня;

5. языки программирования низкого уровня ("воспринимаемые" вычислительной машиной).

Часто под изучением программирования понимается быстрое овладение навыком использования языковых конструкций того или иного языка программирования. Но от программиста требуются более фундаментальные знания. Мы ставим своей целью изучение программирования, а не конкретного языка программирования, хотя, конечно, без знания языка программирования нам не обойтись.

Рассмотрим этапы решения задачи на ЭВМ:

1. Постановка задачи: формулируется цель задачи (на естественном языке) и рассматривается применимость компьютера для решения этой задачи. Важно на этом этапе получить осознание цели задачи, в чем могут помочь различные примеры;

2. Математическая формализация задачи: составляется математическая модель (система уравнений, рекуррентные соотношения и т.д.);

3. Составляется алгоритм решения задачи;

4. Написание программы;

5. Отладка.

1.2 Разработка алгоритма

Определение 1.1 Алгоритм – *формально описанная последовательность действий, позволяющая за конечное время решить поставленную задачу, т.е. преобразовать входные данные в конечный результат.*

Алгоритм должен обладать следующими качествами:

- массовость – применимость к возможно большему кругу однотипных задач;

- дискретность – возможность разделения алгоритма на определенное число шагов;

- детерминированность – на каждом шаге алгоритма получаем результат, который однозначно вытекает из результатов на предыдущих шагах и входных данных;
- конечность – время выполнения алгоритма должно быть конечным;
- результативность – в результате работы алгоритма должен быть получен желаемый результат.

Алгоритм может быть записан разными способами

- не естественном языке;
- графическим способом (блок-схемы, структуры, ...);
- на языке программирования.

Определение 1.2 Алгоритмический язык – *это язык, т.е. последовательность слов, удовлетворяющая определенным синтаксическим правилам, позволяющая описывать алгоритм.*

Определение 1.3 Программирование – *процесс написания алгоритма (программы) на том или ином алгоритмическом языке.*

1.2.1 Основные этапы разработки алгоритма

Первым этапом при разработке алгоритма является этап **анализа задачи**. На данном этапе необходимо:

- установить, что является входом и выходом будущего алгоритма;
- выделить необходимые основные отношения между входными и выходными объектами и их компонентами;
- выделить подцели, которые необходимо достичь для решения задачи;

Результатом данного этапа должна быть *спецификация алгоритма*, т.е. формулировка в самом общем виде того, что должен делать алгоритм, чтобы переработать входные данные в выходные.

В формулировках задач, как правило, отсутствуют какие-либо ограничения на размер входных данных. Однако ресурсы вычислительной системы не беспредельны в смысле объема памяти, имеющейся для хранения программы и ее данных и времени работы программы. Поэтому в спецификации алгоритма обязательно должен быть описан (через систему ограничений) класс тех входных данных, которые могут обрабатываться данным алгоритмом. Например, для алгоритмов обработки текстов кроме точного описания структуры текстов в спецификации должна присутствовать и максимально допустимая длина входного текста и т.д.

Второй этап – собственно разработка алгоритма: последовательности действий, которые преобразуют входные данные в требуемый результат. Разработка алгоритма, а также его обоснование и, если необходимо, модификация, существенно осложняется, если разработчик не придерживается с самого начала некоторой стратегии, позволяющей на каждом этапе разработки четко выделять необходимые подцели и прослеживать взаимосвязь между ними. Одной из таких стратегий является разработка алгоритма *"сверху вниз"*. Суть этого метода состоит в том, что алгоритм разрабатывается сверху вниз, начиная со спецификации, полученной на этапе анализа задачи. На каждом этапе принимается небольшое число решений, приводящих к постепенной детализации управляющей и информационной структуры алгоритма. Т.о. получается последовательность все более детальных спецификаций алгоритма, приближающихся к окончательной версии программы.

Этот подход позволяет разбить алгоритм на части (модули), каждая из которых решает самостоятельную (как правило, небольшую) подзадачу. Это дает возможность сосредоточить усилия на решении подзадачи, реализуемой в виде отдельной функции. Связи по управлению между подзадачами осуществляется посредством соответствующих вызовов функций, а передача информации между подзадачами осуществляется через параметры функции и глобальные переменные.

Замечание: Другой подход к программированию – "снизу вверх", когда от средств движутся к цели. Начинающие программисты пишут программы именно в этом стиле, т.к. пишут операторы в порядке вычисления их компьютером. Но такой подход применим только для небольших задач. Главная стратегия для решения сложных задач – нисходящее проектирования. Заметим, что технология нисходящего программирования не зависит от конкретного языка программирования.

При разработке алгоритма важно иметь в виду следующие факторы, которые существенным образом влияют на разрабатываемый алгоритм:

- средства, предоставляемые выбранным языком программирования. Например, в языке C++ допускаются рекурсивные функции, а в Фортране их нет. Таким образом, для разных языков программирования имеется возможность разрабатывать разные алгоритмы;

- структура данных, на которую нацелен алгоритм. Этот фактор оказывает значительное влияние на эффективность алгоритма.

- приближенность представления вещественных чисел в памяти машины. Все вычисления с вещественными числами должны использовать уровень точности, задаваемый программистом. Например, для проверки, лежит ли заданная точка на данной прямой, надо подставить координаты точки в уравнение прямой. Точка лежит на прямой, если в результате получается ноль. Однако из-за приближенности представления чисел равенство почти наверняка никогда не получится. Реально приходится считать, что условие выполняется, если результат меньше по модулю некоторого заранее заданного программистом числа (например, 10^{-5}).

Обоснование правильности алгоритма – также весьма важная задача. Строгое математическое доказательство практически не выполнимо, главным образом из-за того, что трудно доказать правильность работы циклов и рекурсивных процедур. Однако демонстрация правильной работы алгоритма на некоторых тестах еще не означает, что он всегда будет работать правильно (поскольку различных комбинаций входных данных, как правило, бесконечно много). Поэтому алгоритм необходимо сопровождать некоторым рассуждением, которое, даже не будучи строгим доказательством, в достаточно полной мере убеждает нас в правильности работы алгоритма. По-видимому, наилучшим реальным подходом к обоснованию алгоритма является его обоснование “по построению” согласно пошаговой схеме разработки алгоритма сверху вниз. Чтобы получить правильный алгоритм, надо следить за правильной детализацией его шагов в ходе такого построения. Однако следует иметь в виду, что если с самого начала выбран неверный подход к решению задачи, даже самая аккуратная детализация исходной спецификации не позволит получить правильный алгоритм.

1.2.2 Запись алгоритма на языке программирования

Языки программирования прошли в своем развитии длинный путь. Сегодня программы записываются на алгоритмических языках, приближенных к естественному языку. Языки программирования делятся на

- *машинно-зависимые* – языки низкого уровня. Рассчитаны для работы на машине данной определенной архитектуры:
  - машинный код,
  - мнемокод, ASSEMBLER;
- *машинно-независимые* – языки высокого уровня:
  - универсальные языки (PL, Pascal, C),
  - специализированные (Фортран, Алгол, Пролог, Кобол, LISP)

Развитие языков программирования:

1. по-видимому, самыми первыми языками программирования можно считать различные тумблеры и кнопки на панели управления некоторого устройства;
2. машинные коды – неудобно, громоздко;
3. символьное кодирование, мнемокод, ASSEMBLER;
4. языки высокого уровня;
5. языки структурного программирования;
6. объектно-ориентированные языки.

Прежде всего, следует отметить, что написание программы представляет собой, по сути, продолжение разработки алгоритма, с дальнейшей детализацией его шагов (в результате чего шаги становятся уже элементарными) и фиксирование его на конкретном языке программирования. Поэтому здесь следует применять тот же принцип нисходящего проектирования, о котором мы говорили выше. Этот метод ориентирован на получение хорошо структурированных программ, структура текста которых отражает структуру вычислений. Для того, чтобы добиться такого эффекта, рекомендуется на всех этапах уточнения использовать структуры языка,

имеющие один вход и один выход (последовательные структуры, условные операторы, операторы выбора, операторы цикла). Следует ограничить употребление оператора перехода. Такая программа называется структурной, она легко читается и понимается не только самим автором, но и другими программистами.

Также для повышения наглядности программы рекомендуется форматировать программу, выделяя при помощи сдвигов начала строк содержательно разные части программы и части, относящиеся к синтаксически разным единицам программного кода. Это позволяет не только облегчить чтение программы, но и облегчает поиск синтаксических ошибок, например место нарушения баланса операторных скобок.

Использование комментариев и содержательно осмысленных имен переменных, типов, констант и функций также весьма простой и мощный прием в деле улучшения наглядности программы. Также неразумно помещать в одну строку программы несколько операторов, поскольку это затрудняет чтение и модификацию программного кода.

Методы по улучшению качества программ. Приведем несколько важных приемов, с помощью которых можно добиться улучшения качества программ:

- прежде всего, следует отметить важность использования функций как мощный прием программирования. За счет этого уменьшается программный код, повышается структурированность и надежность программы, облегчается процесс ее написания и отладки;
- если значение некоторого выражения используется в программе несколько раз, целесообразно использовать под это значение вспомогательную переменную, не перевычисляя значение выражения несколько раз;
- выражения, значения которых не меняются при выполнении тела цикла, следует вычислить до цикла и занести соответствующее значение во вспомогательную переменную.

1.2.3 Выбор представления данных

Данные, обрабатываемые алгоритмом, делятся на

- входные;
- выходные;

- промежуточные.

Специфика входных и выходных данных состоит в том, что с ними имеет дело не только алгоритм, но и пользователь программы. Поэтому различают два представления таких данных: *внешнее* и *внутреннее*. Основное требование к внешнему представлению данных – его максимальное удобство, понятность и естественность для пользователя, чтобы он мог достаточно просто подготовить данные для ввода и оценить результат по выходным данным. Например, если в качестве входных данных поступает последовательность координат точек плоскости, то естественно требовать, чтобы сначала шли координаты первой точки, потом – второй и т.д. И наоборот, неестественно требовать, чтобы сначала шли все абсциссы, а потом все ординаты, что потребовало бы от пользователя непростой предварительной работы по упорядочению входной информации в соответствии с этими требованиями. Точно так же недопустим такой вывод полученной информации, который потребует от пользователя нетривиальных усилий по переводу напечатанных данных в понятия исходной постановки задачи. Необходимо помнить о содержательном группировании распечатываемых элементов для удобства их восприятия (например, при выводе матрицы печатать каждую строку матрицы с новой строки).

Далее, необходимо зафиксировать внутреннее представление входных, выходных и промежуточных данных, обрабатываемых в программе, т.е. выбрать те средства программирования, которые будут в данном случае использоваться (типы, структуры данных). Отметим, что для входных данных это представление не обязано совпадать с внешним представлением (например, для N точек плоскости можно использовать матрицу размера $N \times 2$, где i – строка задает координаты i-ой точки). Выбор внутреннего представления данных определяется главным образом вопросами эффективности того алгоритма, которые нужно построить для решения задачи, и может оказать существенное влияние как на объем требуемой памяти, так и на время работы алгоритма.

1.2.4 Сложность алгоритма

Часто для одной и той же задачи может быть предложено несколько различных алгоритмов. Естественно, среди всех возможных алгоритмов предпочтение следует отдать лучшему. Какой критерий следует использовать для сравнения алгоритмов друг с другом? (Мы не берем в рассмотрение сложность по разработке алгоритма

и записи его на алгоритмическом языке.) Существуют различные меры сложности алгоритма, в соответствии с которыми один алгоритм может сравниваться с другим. Как правило, любая сложностная характеристика алгоритма оценивается как некоторая функция от размера задачи. При этом важно, как быстро растет эта функция с ростом размера задачи. Под размером задачи обычно понимается некоторая характеристика размера входных данных. Например, для задачи умножения матриц – это размер перемножаемых матриц. Для сортировки последовательности – количество элементов в последовательности.

- *временная сложность* алгоритма оценивает время работы алгоритма. Может оцениваться как количество операций в алгоритме.

- *пространственная сложность* алгоритма оценивает объем требуемой памяти для работы алгоритма. Может оцениваться как количество переменных, используемых в алгоритме.

Говоря об эффективности того или иного алгоритма, прежде всего, имеют в виду время его работы. Однако часто временная и пространственная сложность связаны друг с другом следующим образом. Мы можем уменьшить время работы алгоритма за счет увеличения используемой памяти и, наоборот, при ограниченных ресурсах в памяти увеличивается время работы алгоритма. Всегда следует искать некий компромисс между двумя этими характеристиками.

Мера сложности алгоритма может оцениваться в худшем и в среднем случае. Сложность алгоритма *в худшем случае* – это его сложность для таких входных данных, для которых алгоритм работает дольше всех или требует больше всего памяти, соответственно. Также важной характеристикой является сложность алгоритма *в среднем*, т.е. для большинства случаев входных данных.

1.2.5 Отладка

После написания программы на языке программирования ее вводят в память машины. Программа хранится в оперативной памяти, ее выполняет процессор. Сначала программу требуется перевести с языка высокого уровня в машинные коды. Этот перевод называется **трансляцией**. Осуществляет эту работу специальная программа, которая называется транслятор.

Трансляторы бывают двух видов:

- компиляторы (C++) – переводят текст программы целиком, создавая объектный код;
- интерпретаторы (Бейсик) – переводят построчно и сразу же выполняют строку.

Кроме перевода текста программы в машинные коды транслятор выполняет следующие функции:

- поиск синтаксических ошибок в тексте программы;
- выдача информации об ошибках пользователю;
- при отсутствии ошибок – перевод в машинный код.

Возможные ошибки, возникающие при программировании:

- синтаксические ошибки – определяются на этапе трансляции;
- ошибки выполнения – связаны с недопустимыми результатами работы тех или иных операторов: деление на ноль, логарифм отрицательного числа и т.д. В результате таких ошибок возникает прерывание выполнения программы. Сведения о таких ошибках выдаются пользователю на этапе выполнения.
- логические ошибки. Логические ошибки возникают, как правило, по следующим причинам:
  - во-первых, задача может быть неправильно понята, может оказаться, что решена совсем не та задача, которая была поставлена;
  - неверным может оказаться первоначальный замысел алгоритма, его спецификация;
  - иногда из-за неверно понятого условия задачи выбранный вроде бы верный алгоритм дает правильное решение не для всех входных данных, а только для их узкого подмножества;
  - при разработке и кодировании алгоритма в программу могут быть привнесены разного рода ошибки, например, за счет неверного понимания смысла используемых языковых конструкций;

  Поиск логических ошибок осуществляется программистом. Приемы, которые могут здесь использоваться:

– с помощью тестов (задача запускается для входных данных с заранее известным результатом). При этом в набор тестов необходимо включать входные данные с различными условиями таким образом, чтобы иметь возможность протестировать различные ветви программы;

– вставка в циклические и ветвящиеся структуры отладочной печати, позволяющей проконтролировать состояние переменных в узловых точках программы;

– использование встроенных в систему средств (debug) с возможностью пошагового выполнения программы, вставка контрольных точек и т.д.;

– испытание программы в “экстремальных” ситуациях;

– упрощение записи сложных формул, разбитие их на последовательность простых.

1.2.6 Эксплуатация программы

При завершении создания программного продукта на него готовятся документы, облегчающие работу с ним, а также в случае необходимости возможную его модификацию. В данный пакет документов, как правило, включаются:

- пояснительная записка;
- описание программного проекта (структура программы, описание используемых структур данных, алгоритмов, блок/схемы);
- текст программы;
- методика испытаний программы, набор тестов;
- руководство программисту.

1.3 Блок-схемы

Одним из способов графической записи алгоритма является запись алгоритма в виде блок-схем.

Определение 1.4 Блок-схема – *это графическая запись алгоритма, состоящая следующих элементов: операторные блоки, изображаемые в виде прямоугольников, и условные блоки, изображаемые в виде ромбов, а также два специальных элемента – "н" (начало) и "к" (конец). Элементы в блок схеме соединены стрелками. Каждый операторный элемент может иметь несколько входных стрелок и ровно одну выходную стрелку. Условный блок может иметь несколько входных стрелок и две выходные стрелки, помеченные + и −. Из элемента "н" выходит одна стрелка, не входит ни одной. Из элемента "к" не выходит ни одной стрелки.*

Семантика: Пусть S – операторный блок. После операторного блока S выполняется блок, на который указывает единственная выходящая из S стрелка. Пусть B – условный блок. После выполнения блока B выполняется блок, в который ведет выходящая из B стрелка, помеченная $+$, если условие, записанное в B истинно, и выполняется блок, в который ведет выходящая из B стрелка, помеченная $-$, если условие, записанное в B ложно. Первым выполняется блок, на который указывает стрелка, выходящая из "н". Выполнение прекращается по достижению блока "к".

Блок-схемы хорошо описывают *логическую структуру* выполнения алгоритма. Объектами логической структуры являются действия или операторы. Связями элементов логической структуры являются связи между операторами, оформленные в виде стрелок.

Структурные блок-схемы – блок-схемы, составленные композицией следующих типов базовых конструкций:

- последовательная структура;
- условная структура;
- циклическая структура.

Не менее важна *информационная структура* алгоритма. Объектами информационной структуры алгоритма являются переменные, а связи указывают на наличие зависимости вычисления одних переменных через другие. Из блок-схемы информационная структура совершенно не видна, хотя играет при разработке алгоритма не менее важную роль. Перед тем, как описывать алгоритм, необходимо продумать информационную структуру:

- какие данные в каких переменных будем хранить;
- какие переменные через какие будем вычислять.

1.4 Структурное программирование

В теории программирования доказано, что любую программу можно написать с использованием трех основных базовых конструкций:

- составного оператора;
- ветвления;
- цикла.

Эти три конструкции называют базовыми конструкциями структурного программирования. Особенностью данных конструкция является то, что все они имеют только один вход и один выход. Конструкции могут вкладываться друг в друга произвольным образом. Структурное программирование часто называют программированием без goto. Написанные в структурном стиле программы легче читаются, их легче отлаживать, яснее прослеживается основной алгоритм.

Структурная блок-схема – блок-схема, представляющая собой одну из базовых конструкций, подконструкциями которой являются в свою очередь структурные блок схемы.

1.4.1 Методы структурирования блок схем

Ниже приведены некоторые метода, которые позволят преобразовать неструктурную блок-схему в структурную:

- дублирование действий;
- единая точка входа;
- комбинирование действий;
- использование флажков.

  Флажок – переменная, принимающая конечное множество значений (как правило, 2 значения) и фиксирующая состояние алгоритма.

Определение 1.5 Алгоритмический язык – *это язык, т.е. последовательность слов, удовлетворяющая определенным синтаксическим правилам, позволяющая описывать алгоритм.*

Определение 1.6 Программирование – *процесс написания алгоритма (программы) на том или ином алгоритмическом языке.*

Точно так же, как для любого языка существуют буквы, слова, словосочетания и предложения, в языке C++ можно выделить

- символы языка, составляющие его алфавит. Это неделимые знаки, с помощью которых пишутся все тексты на языке;
- лексемы – минимальные единицы языка, имеющие самостоятельный смысл. К лексемам относятся идентификаторы, ключевые слова, знаки операций, константы, разделители;
- выражения, задающие правило вычисления некоторого значения;
- операторы, задающие законченное описание некоторого действия.

Идентификатор – последовательность букв или цифр, начинающаяся не с цифры. Также в идентификаторе может содержаться знак подчеркивания. Идентификатор задает имя программного объекта. Мы уже упоминали, что используемые имена всегда лучше делать мнемоничными.

Ключевые слова – зарезервированные языком идентификаторы, которые имеют специальный смысл.

Комментарии – последовательность любых символов, начинающаяся с // или расположенная между символами /* и */. Комментарии игнорируются компилятором. Вложенные комментарии не допускаются.

В науках о языке (лингвистике, математике) понятия, относящиеся к видимому (форме определения, обозначениям, именам объектов) принято относить к синтаксису. Понятия, относящиеся к смыслу, к подразумеваемому – к семантике, а к вопросам оптимального использования языка – к прагматике. Изучая язык программирования, будем использовать данные понятия. Для любой конструкции языка
Синтаксис – как это записывается;
Семантика – что это означает;
Прагматика – как это используется.

1.5 Типы данных

Поскольку все данные, которые обрабатывает программа, хранятся в оперативной памяти в двоичном типе, чтобы с какими данными в данный момент производится работа, введена концепция типов данных. Тип данных определяет:

- формат представления данных в памяти машины;
- диапазон допустимых значений для величин данного типа (диапазон допустимых значение определяется форматом представления данных данного типа);
- множество допустимых операций над величинами данного типа.

Все типы данных можно классифицировать следующим образом. Во-первых, типы можно подразделить на

- простые (неделимые);
- составные (определяемые поверх простых типов данных).

Также можно разделить типы данных на

- стандартные (входящие в стандарт языка);
- пользовательские (определяемые пользователем с использованием средств языка).

Простые типы данных – это типы **bool**, **int**, **float**, **double**, **char**. Существуют спецификаторы, которые уточняют представление простых типов данных:

- short;
- long;
- signed;
- unsigned.

Целый тип int. Переменная данного типа занимает в памяти машины 4 байта (для 32-разрядного процессора) и хранится в двоичном виде. Старший бит отводится под знак числа (0 – положительное, 1 – отрицательное). Возможно использование спецификаторов (short – 2 байта). Дипазон допустимых значений определяется предельно представимыми числами в соответствующем количестве байтов. Основные операции над величинами данного типа: арифметические операции +,-,*,/ (частное от деления),% (остаток от деления), операции сравнения >,<,>=,<=,== (проверка на равенство), != (проверка на неравенство), операция присваивания, ++ (операция увеличения на 1), – (операция уменьшения на 1).

Вещественные типы float и double – типы данных для вещественных чисел, представленных в формате с плавающей точкой. Представление числа в формате с плавающей точкой означает, что число представляется в виде двух чисел – *мантиссы* и *порядка*. Под величины типа **float** отводится 4 байта, под величины типа **double** отводится 8 байтов. Как правило, наиболее употребим тип **double**, имеющий большую точность представления чисел, тем более что большинство библиотечных функций выполняют вычисления именно с типом **double**. Из 64 бит, отводимых под значение типа **double** один бит отводится под знак числа, 11 – под порядок, 52 – под мантиссу. Само число определяется как мантисса * 10 в степени порядка. Знак числа определяется знаковым битом. Порядок определяет диапазон допустимых значений, мантисса – точность представления числа. Возможно применение спецификатора к типу **double**, обозначающее большую точность (занимает 10 байтов). Основные операции над величинами данного типа записываются практически так же как у целочисленного типа. Исключение составляет операция деления /, возвращающая вещественное число – частное от деления.

Логический тип bool. Переменная данного типа занимает в памяти машины 1 байт и может принимать только 2 значения: true и false. Внутреннее представление для true – единица, для false – ноль. Основные операции над величинами данного типа: && (логическое “И”), || (логическое “ИЛИ”), ! (операция отрицания), == (проверка на равенство). Все операции сравнения над величинами некоторых типов имеют результат логического типа. Ниже приведены таблицы истинности для основных логических операций:

a	!a
0	1
1	0

a	b	a==b
0	0	1
0	1	0
1	0	0
1	1	1

a	b	a||b
0	0	0
0	1	1
1	0	1
1	1	1

a	b	a&&b
0	0	0
0	1	0
1	0	0
1	1	1

Алгоритмическое определение булевских операций. Как видно из таблиц, результат логической операции “И” равен false, если хотя бы один из аргументов равен false, и true в противном случае. Результат логической операции “ИЛИ” равен true, если хотя бы один из аргументов равен true, и false в противном случае. Хотя данные операции коммутативны, алгоритмически они выполняются слева направо. При этом если значения первого операнда достаточно для получения результата, второй операнд не вычисляется. Это достаточно тонкий момент, так как значение второго операнда может быть неопределено и в этом случае порядок следования операндов в этих операциях важен.

Символьный тип char. Переменная данного типа занимает в памяти машины 1 байт, в котором хранится в двоичном виде код символа в некоторой кодировке символов. Операции, производимые над переменными данного типа, соответственно производятся над кодами символов.

1.6 Переменные

Определение 1.7 Переменная – *именованная область памяти, служащая для хранения значений определенного типа.*

Любая переменная должна быть описана до ее использования. При описании переменной

- задается имя переменной – произвольный идентификатор, не совпадающий с ключевыми словами языка;
- определяется тип переменной.

При этом под переменную выделяется область памяти в размере согласно с типом переменной. С этой область памяти связывается имя переменной, так что всюду, где в программе производятся действия с этой переменной, производится обращение именно к этой области памяти.

Например:

```
int x,y;
double z;
```

Объявление переменной с инициализацией позволяет при объявлении переменной сразу задать ей начальное значение.

```
int x=10;
```

либо

```
int x (10);
```

Переменные в программе подразделяются на

- локальные переменные функции (описаны внутри какой-либо функции (например, внутри функции main));
- глобальные переменные (описаны вне функций).

Глобальные переменные доступны в любой функции программы. Локальные переменные доступны внутри блока, в котором они описаны. *Область действия* идентификатора – это часть программы, в которой его можно использовать для доступа к связанной с ним области памяти. Имя переменной должно быть уникальным внутри своей области действия.

1.7 Операторы ветвления

Часто в программе необходимо выполнять различные действия в зависимости от некоторого условия. В этом случае используются так называемые операторы ветвления: условные операторы и операторы выбора.

1.7.1 Условные операторы

Полный условный оператор используется в случае, если необходим выбор последующего действия из двух возможных альтернатив в зависимости от выполнения некоторого условия.

Синтаксис:

if (*условие*)
  оператор_1;
else
  оператор_2;

Семантика: Если (*условие*) истинно, выполнять *оператор_1*, в противном случае выполнять *оператор_2*.

Пример: Ввести с клавиатуры два целых числа a и b. Переменной max присвоить значение наибольшего из введенных чисел.

```
int a,b,max;
cin> >a> > b;
if (a>b)
   max=a;
else
   max=b;
cout> > max;
```

Неполный условный оператор реализует альтернативу "выполнять – не выполнять" оператор.

Синтаксис:

if (*условие*)
   оператор;

Семантика: Если (*условие*) истинно, выполнять *оператор*, в противном случае – ничего не делать.

Пример: Ввести с клавиатуры два целых числа a и b. Переменной max присвоить значение наибольшего из введенных чисел

```
int a,b,max;
cin> >a> > b;
max=b;
if (a>b)
   max=a;
cout> > max;
```

Причины возможных ошибок:

- условие всегда записывается в круглых скобках;
- в соответствии с синтаксисом, после слова **if** может стоять только один оператор (аналогично и для оператора *оператор_ 2*). Если необходимо выполнить

последовательность операторов, необходимо использовать составной оператор, т.е. заключать эту последовательность операторов в операторные скобки;

- точка с запятой ставится после любого оператора, в том числе и после оператора *оператор_1* (т.е. перед словом **else**). В этом заключается отличие от некоторых других языков программирования, в частности, от языка Паскаль, где перед **else** точка с запятой не ставится.

1.7.2 Оператор выбора

Условный оператор предоставляет возможность реализации выбора из двух возможных альтернатив. Часто необходимо производить выбор из большего количества вариантов последующей обработки. Оператор выбора **switch** служит для реализации этой возможности.

Синтаксис:

switch (*выражение*)
{
  case *константное_выражение_1*:*список_операторов_1*;
  case *константное_выражение_2*:*список_операторов_2*;
...
  case *константное_выражение_k*:*список_операторов_k*;
  default:*список_операторов_k+1*;
}

Семантика: Вычисляется значение выражения (*выражение*), записанного после слова **switch** (переключить). Это выражение должно быть порядковым. Полученное значение последовательно сравнивается со значениями (они должны все быть разными) константных выражений, записанных после слов **case** (случай). Как только значение некоторого константного выражения совпало с вычисленным, управление передается первому оператору из списка операторов, помеченного этим константным выражением. После выполнения всех операторов списка, последовательно выполняются все операторы, идущие ниже в операторе **switch**. Для выхода из оператора **switch** следует использовать оператор выхода **break**. Если значение ни одного константного выражения не совпало с вычисленным, управление пере-

дается первому оператору из списка, помеченного словом **default** (по умолчанию). Строка с ключевым словом **default** может отсутствовать.

Пример: Ввести с клавиатуры целые числа a, b и символ знака операции из множества {+, -, *, /, %}. Вывести на экран результат соответствующей операции над аргументами a и b.

```
♯ include<iostream.h>
void main()
{
int a,b,res;
char op;
bool f=true;
cout< <"введите a и b:"
cin> >a> > b;
cout< <"введите знак операции:"
cin> > op;
switch (op)
{
  case '+': res=a+b; break;
  case '-': res=a-b; break;
  case '*': res=a*b; break;
  case '/': res=a/b; break;
  case '%': res=a%b; break;
  default : cout< < "Операция неизвестна."; f=false;
}
if (f) cout< <"Результат= ≪ < res;
}
```

Причины возможных ошибок:

- условие всегда записывается в круглых скобках;
- для выхода из оператора **switch** следует применять оператор выхода **break**, в противном случае выполняются все операторы, записанные ниже. В этом

заключается отличие от некоторых других языков программирования, в частности, от языка Паскаль с его аналогичным оператором **case**.

1.8 Операторы передачи управления

Операторы передачи управления применяются, когда необходимо нарушить естественный порядок выполнения операторов.

1.8.1 Оператор безусловного перехода goto

Синтаксис:

goto *метка*;

Семантика: Должен существовать ровно один оператор, помеченный меткой *метка*. Управление передается на этот оператор. *метка* – это идентификатор, область действия которого – функция, в теле которой он задан.

Использование оператора **goto** нарушает принцип структурного программирования, где любой блок должен иметь один вход и один выход. Поэтому следует избегать использования этого оператора. Использование оператора **goto** оправдано в случаях:

- выход из большой вложенности циклов и переключателей;
- переход из нескольких разных мест программы в одно (например, в конец функции).

1.8.2 Оператор break

Оператор **break** используется внутри операторов цикла, if и swith для обеспечения выхода в точку программу, расположенную непосредственно за оператором, в котором он находится.

1.8.3 Оператор continue

Оператор **continue** используется только внутри операторов цикла. Оператор перехода к следующей итерации **continue** пропускает все оставшиеся операторы теку-

щей итерации цикла и передает управление на начало следующей итерации цикла.

1.9 Операторы цикла

Циклические конструкции языка C++ используются для реализации многократно повторяющихся действий. Любой цикл состоит из следующих элементов:

- установка начальных значений переменных до входа в цикл;
- тело цикла;
- условие продолжения цикла;
- изменение параметров цикла.

Тело цикла – последовательность операторов, входящих в цикл (выполняющихся многократно).

Итерация – один проход выполнения тела цикла.

Параметры цикла – переменные, изменяющиеся в теле цикла и входящие в условие продолжения цикла.

Счетчики цикла – целочисленные параметры цикла, изменяющиеся с постоянным шагом на каждой итерации цикла.

Существование параметров цикла обязательно, иначе (если переменные, входящие в условие продолжения цикла не будут изменяться в теле цикла) условие цикла никогда не перестанет выполняться и цикл будет выполняться бесконечно.

В языке C++ существуют несколько вариантов циклических операторов: **while**, **do while** и **for**.

1.9.1 Оператор цикла while

Синтаксис:

while (*условие*)
  оператор;

Семантика: Пока истинно (*условие*), выполнять *оператор*.

Замечания:

- Отметим, что поскольку выполнение условия проверяется перед телом цикла, то возможна ситуация, когда тело цикла, состоящее из оператора *оператор* ни разу не выполнится (если изначально условие ложно).
- В соответствии с синтаксисом, в теле цикла находится один оператор. Если необходимо выполнять в цикле последовательность операторов, их надо оформлять как составной оператор, заключая эту последовательность операторов в операторные скобки.
- Условие всегда записывается в круглых скобках.

1.9.2 Оператор цикла do while

Синтаксис:

do
оператор
while (*условие*);

Семантика: Выполнять *оператор* пока истинно (*условие*).

Замечания:

- Отметим, что поскольку выполнение условия проверяется после тела цикла, то оператор, составляющий тело цикла хотя бы раз обязательно выполнится. В этом отличие данного оператора цикла от оператора **while**.
- Условие всегда записывается в круглых скобках.

1.9.3 Оператор цикла for

Синтаксис:

for (*выражение_1*;*выражение_2*;*выражение_3*)
оператор;

Семантика:

1. вычисляется значение выражения *выражение_1*;

2. пока истинно условие, задаваемое выражением *выражение_2*,

3. выполняется *оператор*, затем вычисляется *выражение_3*;

Выражение *выражение_1* выполняется один раз. Оно задает установку начальных значений переменных до входа в цикл. Выражение *выражение_3* задает изменение параметров цикла, выполняется после выполнения текущей итерации цикла. Первая и третья группа в круглых скобках может состоять из нескольких выражений. В этом случае они разделяются запятыми. Любая группа в круглых скобках может отсутствовать. Точки с запятой в этом случае обязательны. Отметим, что в соответствии с синтаксисом, в теле цикла находится один оператор. Если необходимо выполнять в цикле последовательность операторов, их надо оформлять как составной оператор, заключая эту последовательность операторов в операторные скобки.

Замечание: Цикл **for** задает компактную запись всех составных элементов, входящих в цикл. Любой цикл **while** может быть приведен к эквивалентному ему циклу **for** и наоборот.

1.10 Понятие потока. Стандартные входной и выходной потоки

Переменные в языке программирование связаны с понятием ячейки памяти. Память подразделяется на внутреннюю (оперативная память) и внешнюю (дисковая память).

Внутренняя память обладает следующими качествами:

- высокая скорость чтения/записи;
- фиксированный объем;
- используется для временного хранения данных в процессе работы программы.

Внешняя память характеризуется следующими особенностями:

- низкая скорость чтения/записи;

- хранение больших объемов информации;
- предназначена для “постоянного” хранения данных (между периода работы программы).

Поскольку переменные хранятся во внутренней памяти только во время работы программы, данные, которые программа может использовать на входе, в общем случае берутся из внешней памяти и результаты работы программы записываются во внешнюю память. То есть программа читает данные из файла и записывает результаты в файл. Кроме того, программа может брать и записывать из внешних устройств, таких, как клавиатура, экран дисплея, принтер, и т.д. В языке программирования C++ одним из способов описания процесса передачи данных в программу и из программы является использование такого понятия, как *поток*.

Определение 1.8 Поток – *абстрактное понятие, относящееся к переносу данных от источника к преемнику.*

По направленности потоки можно разделить на входные, выходные, двунаправленные.

1.10.1 Буферизация ввода/вывода

Обмен данными с внешней памятью и внешними устройствами характеризуется низкой скоростью. Для увеличения скорости передачи данных обмен с потоком производится через буфер.

Определение 1.9 Буфер – *специальная область оперативной памяти фиксированного размера.*

С каждым потоком связан свой буфер. Программа вводит данные из буфера и записывает данные в буфер. Фактическая передача осуществляется при выводе при заполнении буфера (при вводе – при опустошении буфера). Таким образом, вместо многочисленных медленных операций ввода/вывода малых порций данных осуществляются существенно более редкие операции чтения/записи больших порций данных.

При чтении данных в программу из файлов (или записи данных из программы в файл) используют *файловые потоки*. О них в наших лекциях речь пойдет ниже.

Часто программа берет входные с клавиатуры и выводит результаты работы на экран дисплея. Такой ввод/вывод производится через стандартные входные/выходные потоки. Таким образом, стандартный входной поток связан с клавиатурой, стандартный выходной поток связан с экраном дисплея.

Для того чтобы использовать в программе ввод с клавиатуры или вывод на экран, необходимо подключить стандартную библиотеку

♯ include<iostream.h> Название заголовочного файла: i – input (вход), o – output (выход), stream (поток).

1.10.2 Стандартный входной поток.

Стандартный входной поток **cin** (от слова input – вход) связан с клавиатурой и служит для переноса данных с клавиатуры в программу.

Как правило, данные в стандартном входном потоке представлены в символьном виде. Данные разделены символами-разделителями (пробел, перенос строки, знаки табуляции). Также существует специальный символ, обозначающий конец потока. Если при очередном считывании из потока считанный элемент – символ конца потока, это означает, что поток исчерпан (считывать больше нечего). Чтение данных из стандартного входного потока осуществляется следующим образом:

Синтаксис:

cin$>>$*name*;

- **cin** – имя стандартного входного потока;
- *name* – имя переменной, описанной выше в программе.

Семантика:

Пусть $< a_1, a_2, \ldots, a_m >$ – содержимое входного потока, где $< a_1, a_2, \ldots, a_m >$ – элементы, разделенные символами-разделителями. Тогда после операции ввода

- значение a_1 приводится к типу переменной *name*;
- если приведение типа произошло успешно, значение переменной *name* становится равным a_1, в противном случае выдается сообщение об ошибке;
- содержимое потока становится $< a_2, \ldots, a_m >$.

Константа с именем EOF равна символу конца потока. Проверка на конец потока может быть осуществлена проверкой, является ли очередной считанный элемент символом конца потока. Либо использованием специальной функции для потока cin:

Синтаксис функции проверки конца стандартного входного потока:

cin.eof()

Семантика:

$$\textbf{cin}.eof() = \begin{cases} TRUE, & \text{если последний считанный элемент является} \\ & \text{символом конца потока} \\ FALSE, & \text{в противном случае} \end{cases}$$

Поскольку ввод буферизован, помещение в буфер ввода производится после нажатия клавиши перевода строки. Это дает возможность исправлять набранные на клавиатуре символы до того, как нажата клавиша ENTER. Поступление данных из буфера в программу производится при выполнении команды ввода (cin$>>name$;).

Расширенная команда ввода.

cin $>> name_1 >> name_2 >> \ldots >> name_k$;
аналогична последовательности команд

cin$>> name_1$;

cin$>>name_2$;

...

cin$>>name_k$;

Пример: с клавиатуры вводится последовательность целых чисел. Определить количество введенных чисел. Программа приведена на рис.1.1

```
#include <iostream.h>        //подключаем библиотеку для работы
                             // со стандартными входными/выходными потоками

void main()
{
 int x, kol=0;               // x - куда будем считывать
                             // kol - количество считанных элементов
 cin>>x;                     //  считываем первый элемент
 while (!cin.eof())          // пока считанный элемент
                             // не является символом конца потока
 {
  kol=kol+1;
  cin>>x;                    // считываем очередной элемент
 }
 cout<<kol;
}
```

Рис. 1.1: Подсчет количества введенных с клавиатуры чисел

1.10.3 Стандартный выходной поток.

Стандартный выходной поток cout (от слова output – выход) связан с экраном дисплея и служит для переноса данных из программы на экран.

Так же как и в стандартном входном потоке, данные в стандартном выходном потоке представлены в символьном виде.

Запись данных в стандартный выходной поток осуществляется следующим образом:

Синтаксис:

cout$>>$*выражение*;

- **cout** – имя стандартного выходного потока;
- *выражение* – некоторое выражение.

Семантика:

Пусть $< a_1, a_2, \ldots, a_m >$ – содержимое выходного потока. Тогда после выполнения команды вывода

- содержимое выходного потока становится $< a_1, a_2, \ldots, a_m, x >$,

- где x – значение выражения *выражение*;

Из-за буферизированности вывода, фактический вывод на экран производится при заполнении буфера. Существуют команды сброса буфера, которые мы будем рассматривать в теме файловые потоки.

Расширенная команда вывода.

cout $>>$ *выражение*$_1 >>$ *выражение*$_2 >> \ldots >>$ *выражение*$_k$;
аналогична последовательности команд

cout$>>$ *выражение*$_1$;

cout$>>$*выражение*$_2$;

...

cout$>>$*выражение*$_k$;

Некоторые форматирующие символы:

- символ перевода строки – $'\backslash n'$ (это значение хранится также в константе endl);
- горизонтальная табуляция – $'\backslash t'$;
- вертикальная табуляция – $'\backslash v'$;
- звуковой сигнал – $'\backslash a'$;

Пример: Ввести два целых числа n и m $(n < m)$ и вывести на экран все числа $> n$ и $< m$с. Программа приведена на рис. 1.2.

1.11 Массивы

Массив – производный тип данных. Он строится поверх простых типов. Массивы служат для компактного хранения и быстрого доступа к наборам однотипных элементов.

Определение 1.10 Массив – *набор данных одного типа фиксированного размера.*

```
#include <iostream.h>       //подключаем библиотеку для работы
                            // со стандартными входными/выходными потоками

void main()
{
 int n,m;
 cout<<"number n= ";         // выводим на экран приглашение к вводу числа n
 cin>>n;                      //  вводим число n
 cout<< '\n';                 // вывод символа конца строки
 cout<<"number m= ";
 cin>>m;
 cout<< endl;;                // тоже вывод символа конца строки
 for (int i=n+1; i<m; i++)    // выводим числа с n+1 по m-1
 {                            // выводим число i и пробел
  cout<<i<<' ';                    // выводим число i и пробел

 }
}
```

Рис. 1.2: Работа со стандартным выходным потоком

Для всего этого набора используют одно имя. Элементы в наборе перенумерованы, и доступ к каждому элементу из набора осуществляется при помощи номера (индекса). Нумерация элементов в массиве всегда осуществляется с нуля. При определении массива и при работе с его элементами используются квадратные скобки [].

Синтаксис определения массива:

type name[*size*];

где

- *type* – тип каждого элемента массива (все элементы массива имеют один тип данных);
- *name* – имя массива;
- *size* – размер массива (количество элементов массива.)

При определении массива в программе выделяется память в размере *size* * sizeof(*type*) байтов (по sizeof(*type*) байтов для каждого из *size* элементов). Размер массива не может быть задан переменной, поскольку переменная получает свое

значение на этапе выполнения, а память под массив выделяется на этапе трансляции и компилятор должен знать, какого размера память нужно выделить. Память под массив выделяется сплошным фрагментом. Элементы в этом фрагменте памяти идут по порядку, начиная с элемента с индексом ноль до последнего элемента массива (элемента с номером *size*-1).

Например:

```
const int N=100;
int y[N],a[35];
double z[10];
```

Объявляются массивы: целочисленный массив y, состоящий из N элементов (перенумерованных с 0 до $N-1$), массив из 35 целых чисел a и массив из вещественных чисел размера 10 (последний элемент имеет номер 9).

Размер массива предпочтительнее задавать именованной константой (как это сделано для массива y), так как в случае необходимости изменения размера массива это влечет необходимость модификации только значения константы. В противном случае всюду в программе, где используется явный размер массива, будет необходимо изменять этот размер на другой.

Доступ к элементам массива осуществляется по номеру (который называется *индексом*). Индекс элемента должен задаваться целочисленным выражением, например:

```
y[50]=3;
int i=26;
a[i-1]=y[2*i+1];
z[i-7]=4;
```

При доступе к элементам массива важно следить за корректным значением индекса (которое должно быть ≥ 0 и $\leq$ *size*-1)). Компилятор языка C++ не осуществляет проверку выхода за пределы массива. Это целиком и полностью ложится на плечи программиста. При неправильном значении индекса фактическая операция будет производиться с участком памяти, расположенном за пределами массива и может привести к непредсказуемым последствиям в работе программы.

Массив может быть объявлен с инициализацией. Например:

int a[35]={3,6,7};

объявляется целочисленный массив размера 35. При этом нулевому элементу сразу присваивается значение 3, первому – 6, второму – 7, остальные элементы дополняются нулями.

Если размер массива не указан, то количество элементов в массиве определяется по количеству перечисленных в инициализации значений:

int a (3,6,7);

задает массив длины 3 (размер устанавливается автоматически по количеству перечисленных инициальных значений массива). Массив инициализируется указанными в скобках значениями.

Групповых операций над массивами нет. Все операции должны быть выполнены программным путем над каждым элементом массива.

Имя массива задает адрес начала массива (адрес участка памяти, отведенного под массив).

В функцию массив может передаваться только по ссылке. Поскольку имя массива задает адрес начала массива, то при передаче одномерного массива в функцию в качестве параметра размер передаваемого массива в скобках можно не указывать (скобки писать нужно обязательно, они говорят о том, что данный параметр – одномерный массив). Размер массива в этом случае передается отдельным целочисленным параметром.

Итак, подведем итоги:

- Все элементы массива имеют один тип данных;
- Размер массива должен быть задан константой;
- Элементы нумеруются всегда с 0. Индекс последнего элемента равен $N-1$, где N – размер массива;
- Компилятор не осуществляет проверки выхода за пределы массива;

- Корректное значение индекса элемента массива должно лежать в пределах от 0 до $N-1$, где N – размер массива;
- Переменная имя массива содержит адрес участка памяти, выделенной под массив;
- Нет групповых операций над массивами;
- В функцию массив может передаваться только по ссылке.

1.11.1 Многомерные массивы

Тип элементов массива в свою очередь тоже может быть массивом. В этом случае говорят о многомерных массивах.

Например:

```
const int N=100;
int y[N][10];
```

Объявляется массив двумерный y, состоящий из N элементов, каждый из которых является массивом из 10 элементов. Такой массив можно рассматривать как матрицу размера $N \times 10$. Память под такой двумерный массив выделяется сплошным куском, элементы располагаются в следующем порядке:

$$y[0][0], y[0][1], \ldots y[0][9], y[1][0], y[1][1], \ldots y[N-1][9].$$

1.12 Некоторые аспекты математической логики в программировании

В языке программирования C++ с математической логикой связан тип **bool**.

Упрощение булевских выражений. Приведем некоторые полезные формулы упрощения логических выражений.

Логическое выражение – выражение, принимающее булевское значение (TRUE или FALSE).

Пусть a, b – логические выражения. Тогда выполняется

- $!\,(!\,b) = b$;
- $!\,(a \,\&\&\, b) = (!\,a)||(!\,b)$;
- $!\,(a \,||\, b) = (!\,a)\&\&(!\,b)$;

1.12.1 Стратегии вычисления кванторов

Язык математической логики содержит кванторные операции
$\forall$ – квантор всеобщности,
$\exists$ – квантор существования.

Пусть переменная x принимает значения из множества X. Пусть $B(x)$ логическое выражение, зависящее от переменной x. Определим логические выражения, которые назовем *"**A**"- формула* и *"**E**"- формула* и приведем стратегии вычисления данных формул.

"A"- формула: $\forall x\, B(x)$ – для всех x выполняется свойство $B(x)$

Пусть $x \in X = \{x_1, \ldots, x_n\}$, тогда

$$\forall x\, B(x) \;\Leftrightarrow\; B(x_1) \,\&\&\, B(x_2) \,\&\&\, \ldots \,\&\& B(x_n)$$

Можно задать рекуррентное определение квантора:

$b_0 = true$;
$b_{i+1} = b_i \,\&\&\, B(x_{i+1})$

Это определение дает следующую стратегию вычисления *"**A**"- свойства.*

b=true;	предполагаем, что свойство $B(x)$ выполняется для всех x
x = первое x	
while (не перебрали все x && b)	все еще выполняется свойство $B(x)$
{	
if $(!\,B(x))$ then b =FALSE;	если для очередного x свойство $B(x)$ не выполняется, значит наше предположение не верно,

	заменяем b на FALSE и
	выходим из цикла, ответ уже получен, дальше
	ничего проверять не надо,
else $x =$ следующий x;	иначе идем проверять следующий x.
}	
cout$<<$ b;	Значение b – ответ.

"E"- формула: $\exists x\, B(x)$ – существует x, для которого выполняется свойство $B(x)$

Пусть $x \in X = \{x_1, \ldots, x_n\}$, тогда

$$\exists x\, B(x) \;\Leftrightarrow\; B(x_1) \,||\, B(x_2) \,||\, \ldots \,||B(x_n)$$

Можно задать рекуррентное определение квантора:

$b_0 = false$;
$b_{i+1} = b_i \,||\, B(x_{i+1})$

Это определение дает следующую стратегию вычисления *"***E***"- свойства.*

b=false;	предполагаем, что не существует x,
	для которого выполняется свойство $B(x)$
$x =$ первое x	
while (не перебрали все x && ! b)	все еще не нашлось x, для которого
	выполняется свойство $B(x)$
{	
if ($!\, B(x)$) then $b =$TRUE;	если для очередного x свойство $B(x)$ выполняется,
	значит такой x существует, заменяем b на TRUE и
	выходим из цикла, ответ уже получен,
	дальше ничего проверять не надо,
else $x =$ следующий x;	иначе идем проверять следующий x.
}	
cout$<<$ b;	Значение b – ответ.

Примеры задач на вычисление $\exists$ и $\forall$ свойств:

- Проверить упорядоченность массива $x = (x_0, \ldots, x_{N-1})$ в порядке неубывания. Задача может быть сформулирована как проверка $\forall$-свойства следующим образом:
  $(\forall\, i \in \{0, \ldots N-2\} : x[i] \leq x[i+1])$;

- Поиск заданного элемента y в массиве $x = (x_0, \ldots, x_{N-1})$. Задача может быть сформулирована как проверка $\exists$-свойства следующим образом:
  $(\exists\, i \in \{0, \ldots N-1\} : x[i] == y)$;

- Проверить равенство двух массивов $x = (x_0, \ldots, x_{N-1})$ и $y = (y_0, \ldots, y_{N-1})$. Задача может быть сформулирована как проверка $\forall$-свойства следующим образом:
  $(\forall\, i \in \{0, \ldots N-1\} : x[i] == y[i])$;

Свойство $B(x)$, которое проверяется для каждого элемента при вычислении $\exists$- и $\forall$-свойств, в свою очередь могут быть сложными свойствами. В частности, это опять может быть $\exists$- и $\forall$-свойство. В этом случае при проверке свойства для очередного элемента также используем рассмотренную выше стратегию.

Примеры задач на вычисление вложенных $\exists$ и $\forall$ свойств:

- Проверить, входит ли массив $y = (y_0, \ldots, y_{M-1})$ в массив $x = (x_0, \ldots, x_{N-1})$ как сплошной фрагмент ($M < N$). Задача может быть сформулирована следующим образом:
  $(\exists\, i \in \{0, N-M\} : (\forall\, j \in \{0, \ldots, M-1\} : y[j] == x[i+j]))$;

- Проверка периодичности числового массива $x = (x_0, \ldots, x_{N-1})$. Задача может быть сформулирована следующим образом:
  $(\exists\, r \in \{1, \ldots n/2\} : (\forall\, i \in \{0, \ldots, N-r-1\} : x[i] == x[i+r]))$.

1.13 Реализация операций над множествами с использованием массивов

Пусть $S = \{s_0, s_1, \ldots, s_{n-1}\}$ – некоторое множество. Назовем его *базовым множеством*. Множеством A над S будем называть произвольную совокупность элементов из S. Такие совокупности не считаются упорядоченными. Все множества, различающиеся лишь перестановкой элементов, считаются равными.

Например, пусть $S = \{0, 1, \ldots, 9\}$. Примеры множеств над S:
$A = \emptyset$ – пустое множество,
$A = \{5, 3, 9\} = \{3, 5, 9\} = \{9, 3, 5\}$,
$A = \{1, 3, 5, 7, 9\}$,
$A = \{0, 1, \ldots, 9\}$.
Отметим, что природа элементов базового множества не имеет значения, так как любое множество можно перенумеровать, и использовать вместо самих элементов их номера. Везде далее будем рассматривать множества над базовым множеством S, состоящим из n элементов.

1.13.1 Операции над множествами

Рассмотрим основные операции, используемые при работе с множествами.

Предикаты:

- Принадлежность элемента множеству Пусть x – элемент базового множества, A – множество.

$$x \in A = \begin{cases} TRUE, & \text{если элемент } x \text{ принадлежит множеству A} \\ FALSE, & \text{если элемент } x \text{ не принадлежит множеству A} \end{cases}$$

- Отношения над множествами.

  Пусть A_1, A_2 – множества.

$$A_1 \subseteq A_2 = \begin{cases} TRUE, & \text{если множество} A_1 \text{ является подмножеством } A_2 \\ FALSE, & \text{в противном случае} \end{cases}$$

  Так же существуют отношения $A_1 \subset A_2$, $A_1 == A_2, A_1 \neq A_2$ и т.д.

Операции над множествами:

- Объединение множеств $A_1 \bigcup A_2$;
- Пересечение множеств $A_1 \bigcap A_2$;
- Разность множеств $A_1 \setminus A_2$
- Добавление элемента к множеству;
- и т.д.

1.13.2 Представление множества при помощи массива

Любое множество A над базовым множеством S из n элементов можно однозначно задать при помощи характеристического вектора ($a = (a_0, a_1, \ldots, a_{n-1})$) из нулей и единиц таким, что

$a_i = 1$, если элемент s_i принадлежит множеству A,

$a_i = 0$, если элемент s_i не принадлежит множеству A.

Таким образом, например, вектор, состоящий только из нулей, задает пустое множество, а вектор, состоящих только из единиц, задает множество, совпадающее с базовым множеством S. Тогда можно сопоставить теоретико-множественные операции и операции над характеристическими векторами.

$x \in A \quad \Leftrightarrow a[x] == 1;$
$A \subseteq B \quad \Leftrightarrow \forall x\, a[x] == 1 \Rightarrow b[x] == 1;$
$C = A \bigcup B \quad \Leftrightarrow \forall x\, c[x] = a[x]\, ||\, b[x];$
$C = A \bigcap B \quad \Leftrightarrow \forall x\, c[x] = a[x]\, \&\&\, b[x];$
$C = A \setminus B \quad \Leftrightarrow \forall x\, c[x] = a[x]\, \&\&\, !b[x];$
и т.д.

Используя такое представление множества при помощи характеристического вектора операции над множествами можно реализовать при помощи операций над массивами.

1.13.3 Пример: Алгоритм “Решето Эратосфена”

Рассмотрим следующую задачу: для заданного простого числа n найти все простые числа, не превосходящие n.

Идея алгоритма: Рассмотрим два множества: NUMBERS, содержащее все целые числа от 2 до n, и PRIMES, которое изначально пусто. На каждом шаге алгоритма

- добавляем наименьшее число a из множества NUMBERS во множество PRIMES,
- удаляем из множества NUMBERS число a и все числа, кратные a,
- пока множество NUMBERS не станет пустым множеством.

Таким образом, образно можно представить, что мы как бы просеиваем через решето, удаляя из множества NUMBERS, все числа, которые однозначно не являются простыми, пока решето не окажется пустым.

Осталось запрограммировать данный алгоритм, задавая используемые в алгоритме множества при помощи массивов размерности n, и реализуя соответствующие операции над множествами при помощи операций над массивами.

1.14 Представление полиномов при помощи массивов. Схема Горнера.

Полином от одной переменной степени $N-1$

$$P(x) = p_0 + p_1 * x + \cdots + p_{N-1} * x^{N-1}$$

однозначно задается набором своих коэффициентов $p_0, \ldots, p_{N-1}$. Поэтому любой полином степени $< N$ можно задать при помощи массива его коэффициентов размерности N

$$P = (p_0, p_1, \ldots, p_{N-1}$$

Если фактическая степень полинома $< N-1$, то коэффициенты при старших степенях x равны нулю.

Среди операций над полиномами можно выделить следующие:

- сложение полиномов.
- вычитание полиномов.
- умножение полинома на число.
- умножение двух полиномов.
- взятие производной
- взятие первообразной.
- вычисление полинома в точке $x = a$.

Используя представление полиномов при помощи массивов коэффициентов, операции над полиномами сводятся к операциям над массивами, их задающими. Например, сложение полиномов. Результат этой операции – третий полином, получающийся в результате суммы двух полиномов. Поскольку при сложении полиномов коэффициенты при одинаковых степенях x складываются, данная операция сводится к покомпонентному сложению двух массивов. При умножении полинома на число a все коэффициенты полинома умножаются на a. Ниже мы более подробно рассмотрим операцию вычисления полинома в точке $x = a$ и приведем два алгоритма решения данной задачи.

1.14.1 Вычисление полинома в точке. Схема Горнера

Постановка задачи следующая: полином $P(x)$ задан массивом своих коэффициентов $P = (p_0, \ldots, p_{N-1})$. Для заданного числа a найти значение полинома $P(x)$ в точке $x = a$.

Прямой алгоритм $Alg1$ вычисления полинома в точке основан на вычислении, основанном на естественной записи полинома. Каждый шаг алгоритма $Alg1$ состоит из вычислении очередной степень x, домножении его на соответствующий коэффициент полинома и прибавления полученного результата к сумме. Фрагмент программы, реализующей данный алгоритм, приведен на рис. 1.3. Сложность алгоритма пропорциональна $3 * N$ (три перечисленные выше операции на каждый коэффициент полинома).

Схема Горнера для вычисления полинома в точке.

Представим полином в виде

$$P(x) = p_0 + x * (p_1 + x(p_2 + \cdots + x(p_{N-2} + x * p_{N-1}) \ldots).$$

Используя такое представление полинома можно предложить другой алгоритм $Alg2$ вычисления вычисления полинома в точке, который называется схемой Горнера. А именно, вычисление начинаем со старших коэффициентов полинома (с самой внутренней скобки). Каждый шаг алгоритма $Alg2$ состоит из домножения результата на x и последующего прибавления к нему очередного коэффициента. Фрагмент программы, реализующей даный алгоритм, приведен на рис. 1.3. Поскольку на каждом шаге данного алгоритма выполняются только две операции (умножение

```
#include <iostream.h>
void main()
{
 const int N=50;    //степень полинома
 double p[N];       //полином
 double res, x, a;
 //...              вводим полином
 cin>>a;
 x=1;
 res=p[0];
 for (int i=1; i<N; i++)
 {
  x=x*a;
  res=res+x*p[i];
 }
 cout<<"res= "<<res;
}
```

Рис. 1.3: Вычисление полинома в точке. Прямой алгоритм

и сложение), сложность данного алгоритма пропорциональна $2*N$. То есть данный алгоритм эффективнее прямого алгоритма $Alg1$. Кроме того, в данном алгоритме не используется дополнительная переменная для вычисления промежуточной степени x. То есть, алгоритм $Alg2$ эффективнее алгоритма $Alg1$ и по времени и по памяти. Известно, что временная оценка $O(2*N)$ является нижней оценкой для задачи вычисления полинома в точке. Фрагмент программы, реализующей алгоритм $Alg2$, приведен на рис. 1.4.

1.15 Символьный тип данных

Как мы уже говорили, переменные символьного типа занимают в памяти машины один байт, в котором хранится код символа в соответствующей кодировке. Все операции, которые осуществляются над символами, на самом деле выполняются над их кодами.

1.15.1 Преобразования типов из int в char и из char в int

Допустим, мы имеем переменную типа **char**, в которой хранится значение некотрой цифры. Как получить число типа **int**, равное значению этой цифры. Иначе гово-

```
#include <iostream.h>
void main()
{
 const int N=50;    //степень полинома
 double p[N];       //полином
 double res, a;
 //...              вводим полином
 cin>>a;
 res=p[N-1];
 for (int i=N-2; i>=0; i--)
 {
   res=res*a + p[i];
 }
 cout<<"res= "<<res;
}
```

Рис. 1.4: Вычисление полинома в точке. Схема Горнера

ря, как осуществить преобразование данных из **char** в **int**. При выполнении такого преобразования все, что нам необходимо знать, это что коды цифр в любой таблице кодировок идут подряд друг за другом, начиная с кода цифры '0' до кода цифры '9'. Поэтому код цифры '1' ровно на 1 больше кода цифры '0', код цифры '2' ровно на 2 больше кода цифры '0', и т.д. Поэтому чтобы, например, из цифры '7' получить число 7, необходимо из кода цифры '7' (то есть из значения символа '7') вычесть код цифры '0' (то есть значение символа '0').

char c='7';
int x;
x=c-'0';

После выполнения данных команд значение целочисленной переменной $x = 7$;

Обротное преобразование из **int** в **char** выполняется с использованием тех же соображений. Например, код символа '5' ровно на 5 больше кода символа '0', поэтому для получение кода символа 5 необходимо прибавить смещение 5 к коду символа '0'.

int x=5;
char c;
c=x+'0';

После выполнения данных команд значение символьной переменной $=' 5'$;

Пример: Ввести с клавиатуры целое число n и найти сумму его цифр. Программа для данной задачи приведена на рис. 1.5.

```
#include <iostream.h>

void main()
{
 int n, kol, a;
 cin>>n;              // вводим число n
 kol=0;
 while (n>0)
 {
  a=n%10;             //выделяем последнюю цифру
                      //числа n
  kol=kol+(a-'0');    //прибавляем к kol выделенную цифру,
                      //переведя ее предварительно
                      //в тип int
  n=n/10;             //отбрасывает последнюю цифру
                      //из числа n
 }
 cout<<kol;
}
```

Рис. 1.5: Подсчет суммы цифр числа.

1.16 Строки

В языке C++ нет специального строкового типа. Для хранения последовательностей символов используют массивы данных типа char. Поскольку фактическая длина строки может быть меньше размерности выделенного под строку массива, в язык введено соглашение, что строка символов в таких массивах завершается специальным “терминальным ”символом, обозначаемым ’\0’. Такая последовательность символов носит название “си-строка”.

Определение 1.11 *Си-строка – последовательность символов с завершающим терминальным символом* ’\0’.

На соглашение о существовании терминального нуля в конце строки символов рассчитывают все стандартные библиотечные функции обработки строк языка C++. Если на вход таких функций подать строку без терминального нуля в конце, последствия работы такой функции могут быть самыми непредсказуемыми. В остальном правила работы со строками полностью соответствуют правилам работы с массивами. И именно:

- нет групповых операций над символьными массивами (в том числе нет операции присваивания), все операции надо выполнять поэлементно, либо использовать специальные функции библиотеки работы со строками;
- размер символьного массива для хранения строк должен быть константой;
- при описании символьного массива (резервировании памяти под строки) размер массива должен быть, по крайней мере, на единицу больше размера максимально длинной строки, которая будет храниться в этом массиве. Дополнительный элемент необходим под завершающий терминальный ноль;
- компилятор C++ не производит проверку выхода за пределы массива. Этот контроль полностью ложится на плечи программиста.
- в функцию строка (т.е. символьный массив) может передаваться только по ссылке;

Для того чтобы использовать функции библиотеки работы со строками, необходимо подключить заголовочный файл библиотеки:

♯ include<string.h>

Полный перечень функций данной библиотеки можно найти в любом справочнике по C++. Отметим, что в данной библиотеке существуют функции для подсчета длины строки, копирования, сравнения, конкатенации строк, поиск подстроки в строке и многие другие. Однако отметим, что не представляет труда написать любую из этих несложных функции самим.

Отметим также, что при передаче Си-строк в качестве параметров в функции нет необходимости передавать отдельным параметром размер массива, поскольку фактическая длина строки легко определяется по терминальному символу.

Ввод и вывод строк. Ранее мы отметили, что поскольку для хранения Си-строк используются символьные массивы, все правила работы с массивами, естественно переносятся строки. Исключение составляет ввод/вывод строк. Как мы знаем, невозможно ввести или вывести массив одной командой ввода или вывода. Необходимо в цикле вводить каждый элемент массива. Однако ввести строку символов одной командой ввода возможно. А именно:

char buf[100]; объявляем символьный массив, куда будет вводиться строка

cin > > buf; в массив buf вводится последовательность символов
до символа-разделителя (пробел, перевод строки, символы-табуляции).
В конце введенных символов вставляется символ “терминальный ноль”.

Так же одной командой можно вывести Си-строку:

cout < < buf; в выходной поток выводятся символы массива buf,
начиная с символа buf[0] до символа, стоящего
непосредственно перед терминальным символом ’\0’.

Возможные причины ошибок.

- Наиболее распространенная причина ошибок – отсутствие терминального нуля. Как уже упоминалось, все функции библиотеки работы со строками предполагают присутствии символа ’\0’ в конце строки. Отсутствие данного символа естественно не вызовет никаких нареканий компилятора, но последствия работы данных функций будут непредсказуемыми.
- Так же при отсутствии терминального нуля при форматированном выводе строки будут выводиться все символы, пока не обнаружится символ ’\0’. А так как не производится проверки выхода за пределы массива, что в результате окажется в выходном потоке, так же предсказать невозможно.

Пример: Для заданной вводом последовательности слов определить слово максимальной длины. Распечатать длину этого слова и само слово. Фрагмент программы для данной задачи приведен на рис. 1.6.

На рис. 1.7 приведен фрагмент программы для той же задачи с использованием библиотеки string.

```
#include <iostream.h>
void main()
{
 const int N=20;
 char buf[N];                // массив buf для ввода очередного слова
                             // можно ввести максимум 19 символов + один символ для
                             // терминального нуля
 char maxbuf[N];             // массив для слова наибольшей длины
 int l,max;
 max=0;
 cin>>buf;                   // вводим первое слово
 while (!cin.eof())
 {
  l=0;
  while (buf[l]!='\0')       // подсчет длины слова
        l++;
  if (l>max)                 // сравниваем длину текущего слова с max
  {
   max=l;
   for(int i=0; i<=l; i++)   // сохраняем слово, имеющее длину > max
      maxbuf[i]=buf[i];
  }
  cin>>buf;                  // ввод очередного слова
 }
 cout<<endl<<"max length= "<<max<< endl<<"max word: "<< maxbuf;
}
```

Рис. 1.6: Нахождение слово наибольшей длины

```
#include <iostream.h>
#include <string.h>
void main()
{
 const int N=20;
 char buf[N];                // массив buf для ввода очередного слова
                             // можно ввести максимум 19 символов + один символ для
                             // терминального нуля
 char maxbuf[N];             // массив для слова наибольшей длины
 int l,max;
 max=0;
 cin>>buf;                   // вводим первое слово
 while (!cin.eof())
 {
  l=strlen(buf);             // находим длину слова
  if (l>max)                 // сравниваем длину текущего слова с max
       strcpy(maxbuf,buf);   // в maxbuf сохраняем слово, имеющее длину > max
  cin>>buf;                  // ввод очередного слова
 }
 cout<<endl<<"max length= "<<max<< endl<<"max word: "<< maxbuf;
}
```

Рис. 1.7: Нахождение слово наибольшей длины (с использованием функций строковой библиотеки)

1.17 Файлы

Определение 1.12 Файл – *именованная последовательность данных потенциально неограниченной длины, расположенная во внешней памяти.*

Основные особенности файлов:

- потенциально *неограниченный размер* (в отличие, скажем от массивов, где размер изначально фиксирован);
- *последовательный доступ* к компонентам файла (в отличие от массивов с прямым доступам к компонентам).

Логически, любой файл можно трактовать просто как последовательность байт. То, что именно представляет собой эта последовательность байт, трактуется программистом. Мы будем подразделять файлы на текстовые и двоичные (бинарные).

Текстовые файлы – это файлы, в которых информация записана в текстовом (символьном) формате. Если текстовый файл открыть каким-либо текстовым

редактором (например, редактором Блокнот), то информация может быть легко прочитана.

Двоичные файлы – это файлы, в которых информация записана во внутреннем (двоичном) формате соответственно типам данных.

Любой файл хранится в памяти под некоторым именем. Текстовые файлы, как правило, запоминают с расширением ".txt", двоичные – с расширением ".bin".

Считанная из текстового файла информация хранится в программе во внутренней памяти в переменных в соответствующем формате. Поэтому в процессе ввода производится преобразование информации из символьного типа в тип той переменной, куда информация вводится. При выводе данных файл производится обратное преобразование типа из внутреннего формата в символьный. Такой ввод/вывод с преобразованием типов называется *форматированным вводом/выводом* данных.

Ввод/вывод данных без форматирования, как простое перемещение байт из файла в программу или из программы в файл называется *неформатированным вводом/выводом*. Неформатированный ввод/вывод используется при работе с двоичными файлами или когда информация из текстового файла вводится с переменные символьного типа (то есть когда не требуется преобразование типов).

1.18 Файловые потоки

Ранее мы рассматривали стандартные входные/ выходные потоки cin и cout. Если на вход программы данные подаются не с клавиатуры, а из файлов, и выводятся не на экран, а в файл, то в этом случае передача данных производится посредством файловых входных/выходных потоков.

Потоковые файловые функции трактуют ввод/вывод информации в файлы как перемещение потока байтов из программы в файл через буфер оперативной памяти. Файловый буфер имеет определенный по умолчанию размер. При открытии файла создается соответствующий файловый буфер. По завершении работы, файл необходимо закрыть (освободить выделенные ранее ресурсы).

Для того чтобы использовать в программе ввод данных из файла или вывод в файл, необходимо подключить стандартную библиотеку

♯ include<fstream.h> Название заголовочного файла: f – file (файл), stream (поток).

Работа с файлом в программе предполагает следующее.

1. создание потока для данного файла;
2. открытие потока и связывание его с конкретным физическим файлом;
3. чтение/запись;
4. закрытие потока.

1. открытие файла, то есть создать файловый буфер;
2. найти файл на диске;

По завершении работы, файл необходимо закрыть (освободить выделенные ранее ресурсы).

1.19 Текстовые файлы

В данном разделе мы рассмотрим основные команды для работы с потоками для текстовых файлов. Для полного ознакомления с существующими методами обмена с потоками смотри соответствующую литературу.

Текстовые файлы – это файлы, в которых информация хранится в символьном формате. В текстовых файлах могут присутствовать специальные управляющих символы, служащие для форматирования вывода информации (такие, как конец строки, перевод строки, символы табуляции).

Входной файловый поток (текстовые файлы)

1. Создание входного потока (описание переменной входного потока).

   Синтаксис: ifstream *myin*;

   Семантика: Создается переменная с именем *myin* для входного потока (без привязки его к конкретному физическому файлу)

2. Открытие входного файла.

   Синтаксис: *myin*.open(*filename*);

   Синтаксис: во входном потоке *myin* открывается файл для чтения с именем *filename*. При этом на диске должен быть файл с именем *filename*, из которого предполагается считывать данные. Если файл с таким именем не найден,

потоковая переменная *myin* принимает значение 0. Буфер заполняется первой порцией данных из файла. Программа будет брать данные из буфера при очередной команде ввода. При опустошении буфера он будет подгружаться очередной порцией данных из файла. Физическое имя файла *filename* пишется в программе только здесь. Всюду далее в программе к файлу обращаются через имя потока *myin*, поскольку через этот поток поступают данные из файла в программу.

Возможно использование следующей конструкции:

ifstream *myin*(*filename*);

где создается потоковая переменная с одновременным открытием файла для чтения.

3. Форматированный ввод из текстового файла.

   Синтаксис: *myin*> > *name*;

   Семантика: Семантика в точности соответствует семантике соответствующей команды для стандартного входного потока **cin**. Отличие заключается только в имени потока. Также существует команда расширенного ввода.

4. Неформатированный ввод – считывание одного символа.

   Синтаксис: *myin*.get();

   Семантика: Возвращает очередной извлеченный из потока символ.

5. Неформатированный ввод – считывание последовательности символов.

   Синтаксис: *myin*.getline(buf, num, lim='\ n');

   Семантика: Считывает в buf num-1 символов или меньше, если встретится символ lim, включая и сам символ lim. Переменная buf – символьный массив достаточной длины.

6. Пропуск символов.

   Синтаксис: *myin*.ignore(num=1,lim=EOF);

   Семантика: Считывает и пропускает, никуда не записывая, num символов или меньше, если встретится символ lim, включая и сам символ lim.

7. Проверка на конец файла

   Синтаксис: *myin*.eof();

   Семантика: Аналогична семантике соответствующей функции для стандартного входного потока **cin**. Отличие заключается только в имени потока.

8. Опережающий просмотр

   Синтаксис: *myin*.peek();

   Семантика: Функция возвращает следующий символ без удаления его из потока, или EOF, если достигнут конец файла. Применима и для стандартного входного потока **cin**.

9. Закрытие файла

   Синтаксис: *myin*.close();

   Синтаксис: входной файл, связанный с потоком *myin* закрывается.

Выходной файловый поток (текстовые файлы)

1. Создание выходного потока (описание переменной входного потока).

   Синтаксис:

   ofstream *myout*;

   Семантика: Создается переменная с именем *myout* для выходного потока (без привязки его к конкретному физическому файлу)

2. Открытие выходного файла.

   Синтаксис: *myout*.open(*filename*);

   Синтаксис: во входном потоке *myout* открывается файл для записи с именем *filename*. При этом на диске такой файл существует, от очищается. Если такого файла нет, он создается. Физическое имя файла *filename* пишется в программе только здесь. Всюду далее в программе к файлу обращаются через имя потока *myout*, поскольку через этот поток поступают данные из программы в файл.

   Возможно использование следующей конструкции:

ofstream *myout*(*filename*);

где создается потоковая переменная с одновременным открытием файла для записи.

3. Форматированный вывод в текстовый файл.

   Синтаксис: *myout*< < *выражение*;

   Семантика: Семантика в точности соответствует семантике соответствующей команды для стандартного выходного потока **cout**. Отличие заключается только в имени потока. Также существует команда расширенного вывода.

4. Неформатированный вывод – вывод одного символа.

   Синтаксис: *myin*.put(c);

   Семантика: Выводит в поток символ c.

5. Сброс буфера

   Синтаксис: *myin*.flash();

   Семантика: Выводит содержимое потока на физическое устройство (очищает содержимое буфера).

6. Закрытие файла

   Синтаксис: *myin*.close();

   Синтаксис: выходной файл, связанный с потоком *myout* закрывается. При этом содержимое буфера сбрасывается в файл.

Особенности:

- Отметим следующий важный момент. Поскольку при форматированном вводе символы-разделители (пробел, перевод строки, знаки-табуляции) служат для разделения данных друг от друга, применяя такой форматированный ввод невозможно эти символы-разделители ввести в программу (например, нельзя ввести символ пробел). Для ввода этих символов-разделителей надо применять команды неформатированного ввода (get, getline, . . .).

- В отличие от ввода, при выводе символа в выходной поток не имеет значения, какой командой пользоваться: форматированным выводом или неформатированным выводом (put).

- Необходимо закрывать файл после того, как закончена работа с ним. Особенно это касается выходного файла. Поскольку после работы с выходным файлом содержимое буфера сбрасывается в файл при закрытии файла (метод close()), то при отсутствии данной команды возможна потеря последних данных, которые так и не дошли до физического файла. Также сбросить буфер можно при помощи метода flush().

1.20 Линейные упорядоченные последовательности

Определение 1.13 *Последовательность* $a_0, a_1 \ldots, a_{n-1}$ *называется упорядоченной в порядке неубывания, если для любых* $i = 0, \ldots, n-2$ *выполняется* $a_i \leq a_{i+1}$.

В целях простоты изложения под упорядоченностью мы будем понимать именно упорядоченность в порядке неубывания, хотя аналогично можно рассматривать упорядоченность в порядке невозрастания, строгого возрастания или строгого убывания. Также для простоты изложения будем считать, что элементы последовательностей являются целыми числами.

Линейные упорядоченные последовательности чрезвычайно важны для практики, как множество задач, в первую очередь, задачи поиска, для таких последовательностей решаются гораздо более эффективно. В данном разделе мы рассмотрим некоторые алгоритмы на таких последовательностях применительно к разным типам данных (а именно, применительно к разным способам доступа к элементам последовательности – *прямому* и *последовательному*). При хранении последовательности в массиве мы имеем прямой доступ к ее элементам, что существенно для некоторых алгоритмов. Считывание последовательности из входного потока накладывает свои ограничения (а именно, отсутствие повторного доступа к элементам), что требует особых методов. Мы оценим сложность приведенных алгоритмов и сравним ее со сложностью алгоритмов для тех же задач и тех же типов данных для неупорядоченных последовательностей. Сложность алгоритма будем оценивать по количеству операций сравнения в худшем случае. Сложность алгоритма Alg будем обозначать S_{Alg}.

1.20.1 Поиск элемента

Постановка задачи: Пусть дана упорядоченная последовательность $a_0, a_1 \ldots, a_{n-1}$ ($a_0 \leq a_1 \leq \cdots \leq a_{n-1}$) и пусть дан элемент y. Определить имеется ли в последовательность элемент $= y$, то есть проверить свойство

$$\exists i \quad (a_i == y)$$

Поиск элемента в неупорядоченной последовательности. Прежде всего, отметит, что независимо от способа доступа к элементам последовательности при отсутствии упорядоченности нам не остается ничего другого, как просто сравнивать каждый элемент последовательности с элементом y. В худшем случае элемент y может совпасть с последним элементом, или его может просто не оказаться. Количество операций сравнения в худшем случае $= n$, то есть для данного алгоритма (назовем его $Alg1$) выполняется $S_{Alg1} \leq n$.

Поиск элемента в упорядоченном файле. Пусть упорядоченная последовательность $a_0, a_1 \ldots, a_{n-1}$ поступает в программу из входного потока (мы можем считывать данные из файла, вводить с клавиатуры, ...). В отличие от алгоритма для неупорядоченной последовательности, достаточно найти такое первое j, что $y \leq x_j$. Если $y = x_j$, то искомый элемент в последовательности есть, если $y <> x_j$, такого элемента нет. Оценим сложность алгоритма. При наличии элемента y в последовательности сложность алгоритма (назовем его $Alg2$) в худшем случае совпадает со сложностью алгоритма $Alg1$. То есть $S_{Alg2} \leq n$. Однако отметим, что если элемента y в последовательности нет, сложность алгоритма $Alg1$ будет всегда равна n. Для упорядоченной последовательности мы получим результат гораздо раньше.

Дихотомический поиск в упорядоченном массиве. Пусть дан упорядоченный массив $a = (a_0, a_1 \ldots, a_{n-1})$. Требуется узнать, если ли в массиве элемент с заданным значением y.

Наличие прямого доступа к элементам последовательности позволяет использовать гораздо более эффективный алгоритм, чем рассмотренный ранее. Данный алгоритм использует метод, который называется **дихотомия** (или иначе, метод деления пополам), который находит свое применение в различных областях.

Идея метода: На каждом шаге алгоритма (назовем его $Alg1$) делим массив на 2 части, и смотрим, в какой из них может находиться искомое значение. Так до

тех пор, пока либо не найдем элемент y, либо не поймем, что его в массиве нет. Поскольку на каждом шаге размер подмассива, в котором продолжается поиск, уменьшается в 2 раза, количество необходимых шагов в наихудшем случае приблизительно равно $\log n$. Поэтому $S_{Alg3} \leq \log n$.

На рис. 1.8 приведен фрагмент программы, реализующий данный алгоритм. Считаем, что массив уже введен.

```
void main()
{
 const int N=20;
 int a[N];
 int y;
 bool b=false;                // b - результат (нашли элемент y или не нашли)
 int first,last,middle;       // индексы, задающие начало, конец
                              // и середину подмассива
 for(int i=0; i<N; i++)       // вводим массив
         cin>>a[i];
 cin>>y;
 first=0;                     // сначала подмассив, где производим поиск
 last=N-1;                    // совпадает со всем массивом
 while (!b && (first<last)) // пока не нашли y и все еще есть где искать
 {
  middle=(first+last)/2;      // находим середину подмассива
  if (a[middle]==y) b=true; // если элемент на середине совпадает с y - нашли!
  else
      if (y<a[middle]) last=middle-1;    // если y<чем серединный элемент,
                                         //он может находиться только слева
      else first=middle+1;               // если y>чем серединный элемент,
                                         // надо продолжать поиск
                                         // в правой половине
 }
 cout<<b;
}
```

Рис. 1.8: Дихотомический поиск в упорядоченном массиве

1.20.2 Включение одной последовательности в другую

Даны две упорядоченные последовательности $a : a_0, a_1 \ldots, a_{n-1}$ и $b : b_0, b_1 \ldots, b_{m-1}$. Верно ли, что $a \subseteq b$ (a включается в b):

$$\forall i \in [0 \ldots, n-1](\exists j \in [0, \ldots, m-1] : a_i = b_j)$$

В случае неупорядоченных последовательностей для каждого элемента $a_i, i = 0, \ldots, n-1$ нам необходимо выполнять задачу поиска в неупорядоченной последовательности длины m. Сложность данного алгоритма составляет по порядку $n * m$.

Включение одного упорядоченного файла в другой. Если мы имеет дело со структурой данных с последовательным доступом, то для каждого $a_i (i = 0, \ldots, n-1)$ необходимо искать подходящий b_j (как в алгоритме $Alg2$). Переходя к поиску очередного a_i не нужно вставать на начало последовательности b, а продолжать поиск с того места, на котором остановился поиск предыдущего a_i. В результате имеем один проход по каждой последовательности и сложность S данного алгоритма $Alg5$ $S(Alg5) = n + m$. Таким образом, видим, что упорядоченность дает значительное преимущества и для данной задачи.

Включение одного упорядоченного массива в другой. С учетом идей, изложенных в алгоритмах $Alg3$ и $Alg3$ можно предложить соответствующий алгоритм для массивов.

1.20.3 Объединение двух упорядоченных последовательностей

Даны две упорядоченные последовательности $a : a_0, a_1 \ldots, a_{n-1}$ и $b : b_0, b_1 \ldots, b_{m-1}$. Построить упорядоченную последовательность $c : c_0, c_1 \ldots, c_{k-1}$, в которую включить элементы, которые есть или в a или в b. В теоретико-множественном смысле можно сказать, что c есть объединение последовательностей a и b: $c = a \bigcup b$.

В случае неупорядоченных последовательностей в общем-то, задача не имеет большого смысла, поскольку, если мы хотим объединить две неупорядоченные последовательности, получив упорядоченную последовательность, то можно сделать их конкатенацию, а затем отсортировать полученную последовательность. Сложность алгоритма в этом случае совпадает со сложностью алгоритма сортировки. В любом случае для произвольных последовательностей она не линейна.

В случае упорядоченных последовательностей можно предложить следующий однопроходный алгоритм и для массивов и для файлов.

1. Устанавливаемся на начало последовательности a $(i = 0)$, последовательности b $(j = 0)$ и начинаем строить последовательность c $(k = 0)$.

2. Пока не закончатся обе последовательности

3. Если закончилась последовательность a, переписываем текущий элемент из b в c, сдвигаемся в b;

4. Если закончилась последовательность b, переписываем текущий элемент из a в c, сдвигаемся в a;

5. Если ни a ни b не закончились, переписываем в c текущий элемент из той последовательности, где он меньше, сдвигаемся в той последовательности, откуда записали элемент.

6. Переход к шагу 2.

Программа, соответствующая данному алгоритму для массивов, приведена на рис. 1.9

Согласно описанному алгоритму имеем один проход по исходным последовательностям. Поэтому сложность данного алгоритма $Alg6$ линейна $S(Alg6) = n + m$.

Отметим, что при реализации алгоритма важен порядок исследования случаев в цикле: сначала обязательно проверка на окончание последовательностей, а только затем разбор случая, когда не закончились обе последовательности. В противном случае будут анализироваться элементы, находящиеся за пределами массива. Можно предложить слегка видоизмененную модификацию данного алгоритма, когда в цикле вместо операторов if используются операторы while.

1.20.4 Пересечение двух упорядоченных последовательностей

Даны две упорядоченные последовательности $a : a_0, a_1 \ldots, a_{n-1}$ и $b : b_0, b_1 \ldots, b_{m-1}$. Построить упорядоченную последовательность $c : c_0, c_1 \ldots, c_{k-1}$, в которую включить элементы, которые есть и в a и в b. В теоретико-множественном смысле можно сказать, что c есть пересечение последовательностей a и b: $c = a \bigcap b$.

В случае неупорядоченных последовательностей алгоритм имеет сложность $n * m$, поскольку для каждого элемента $a_i, (i = 0, \ldots, n-1)$ необходимо выполнить задачу поиска в неупорядоченной последовательности. При положительном результате поиска элемент a_i записывается в c.

```
#include <iostream.h>
void main()
{
 const int N=20,M=10,L=30;
 int a[N],b[M],c[L];
 int i,j,k;
 // Ввод массивов a и b
 i=0; j=0; k=0;
 while (i<N || j<M)
 {    if (i>=N)
      {  c[k]=b[j];
         j++;
      }   else
            if (j>=M)
            { c[k]=a[i];
              i++;
            }  else
                  if (a[i]>b[j])
                  {  c[k]=b[j];
                     j++;
                  }  else
                     { c[k]=a[i];
                       i++;
                     }
      k++;
 }
 // Вывод массива c
}
```

Рис. 1.9: Объединения двух упорядоченных последовательностей

В случае упорядоченных последовательностей нетрудно разработать линейный алгоритм сложности $n+m$ используя те же соображения, что и для предыдущей задачи.

1.20.5 Разность двух упорядоченных последовательностей

Даны две упорядоченные последовательности $a : a_0, a_1 \ldots, a_{n-1}$ и $b : b_0, b_1 \ldots, b_{m-1}$. Построить упорядоченную последовательность $c : c_0, c_1 \ldots, c_{k-1}$, в которую включить элементы, которые есть в a, но которых нет в b. В теоретико-множественном смысле можно сказать, что c есть разность последовательностей a и b: $c = a \setminus b$.

И для упорядоченных и для неупорядоченных последовательностей соображения аналогичны двум предыдущим задачам.

Таким образом, можно сделать вывод. Многие задачи решаются существенно более эффективно, если исходные данные хранятся в упорядоченном виде. В частности, существуют однопроходные алгоритмы для задач поиска, объединения, пересечения, разности, включения упорядоченных последовательностей. Однако не существует однопроходных алгоритмов сортировки произвольной последовательности. Следовательно, порядок легче сохранять, чем устанавливать.

1.21 Простейшие алгоритмы сортировки последовательности

Сортировка данных – это обработка информации, в результате которой элементы располагаются в определенном порядке в зависимости от значения некоторых признаков элементов, называемых ключевыми. Алгоритмы сортировки находят широкое применение при решении различных задач. Практическая ценность алгоритмов сортировки обусловлена, прежде всего, тем, что поиск, анализ и обработка информации для отсортированных наборов данных выполняются, как правило, гораздо эффективнее, чем для не отсортированных.

В самом общем виде задачу сортировки можно сформулировать следующим образом. Пусть имеется последовательность элементов $R = R_1, R_2, \ldots, R_n$ снабженных, соответственно, ключами $K_1, K_2, \ldots, K_n$. Можно считать, что каждый элемент последовательности $R_i =< K_i, I_i >$, где K_i – это ключ, I_i – информационная часть элемента. В частности, весь элемент может совпадать со своим ключом. В этом

случае часть I может отсутствовать.

Предполагается, что на множестве $\mathcal{K}$ значений, которые могут принимать ключи, введено отношение порядка. А именно, для любых значений $a, b, c \in \mathcal{K}$ выполняется:

1. либо $a < b$, либо $a = b$, либо $a > b$;

2. $a \leq b$ и $b \leq c$ влекут $a \leq c$.

В задаче сортировки требуется переставить элементы $R_1, R_2, \ldots, R_n$ таким образом, чтобы в результирующей последовательности элементы располагались в порядке неубывания значений их ключей, т.е. $K_1 \leq k_2 \leq \cdots \leq K_n$ (где нумерация относится уже к новой последовательности). (Отметим, что аналогично можно рассматривать задачу сортировки по невозрастанию).

В несколько иной интерпретации требуется найти такую перестановку индексов $1, 2, \ldots, n$, что $K_{\pi(1)} \leq K_{\pi(2)} \leq \cdots \leq K_{\pi(n)}$. При этом сами элементы $R_1, R_2, \ldots, R_n$ не перемещаются, т.е. остаются на своих местах. Очевидно, располагая перестановкой π нетрудно при необходимости осуществить и физическую перестановку элементов. В данной интерпретации задача может иметь смысл, когда элементы последовательности R занимают большой объем памяти и требуют много времени на физическое перемещение. Получение решения в виде перестановки индексов π может быть более эффективным по времени.

В качестве примера рассмотрим следующую задачу. Пусть имеется русско-английский словарь. Каждую запись словаря можно рассматривать как пару $< R, A >$, где R – слово на русском языке, A – его перевод на английском языке. Если данный словарь отсортирован по ключу R (A в таком случае можно рассматривать как информационную часть элемента), то им легко пользоваться при переводе текста с английского языка на русский. Если необходимо использовать словарь для перевода текста с английского языка на русский, то словарем будет удобнее пользоваться, если предварительно отсортировать его по ключу A.

В настоящее время разработаны множество различных алгоритмов сортировки. Выбор того или иного алгоритма обуславливается конкретной задачей: типом данных, используемых для сортируемой последовательности, размером последовательности, способом доступа, и различными другими факторами. Ниже мы разберем некоторые простейшие алгоритмы сортировки. Для простоты изложения и без потери общности предполагаем, что сортируемая последовательность R – это

последовательность целых чисел. Будем считать, что сортируемая последовательность задана массивом.

Итак, дан массив целых чисел.

const int N=100;

int a[N];

Требуется переставить элементы массива таким образом, чтобы они шли в порядке неубывания, т.е. чтобы для любых $i = 1, \ldots N-1$ выполнялось $a[i-1] \leq a[i]$.

1.21.1 Алгоритм сортировки методом обмена пар

Алгоритм сортировки методом обмена пар называют еще иногда методом "пузырька".

Алгоритм: Проходим по массиву слева направо и на каждом шаге анализируем пару рядом стоящих элементов $a[i-1]$ и $a[i]$ $(i = 1, \ldots, N-1)$. Если пара стоит неправильно (т.е. $a[i-1] > a[i]$), то переставляем местами эти элементы. После первого прохода по массиву наибольший элемент массива встанет на свое окончательное место (на $N-1$-ую позиция). Его исключаем из рассмотрения и снова запускаем процесс на с подмассиве $a[0] \ldots a[N-2]$. В наихудшем случае потребуется $N-1$ проход по массиву. Длина рассматриваемого подмассива на каждом шаге уменьшается.

Пример: Упорядочить по возрастанию массив (7, 5, 6, 2, 3, 1). На примере 1 играет роль пузырька.

7	5	6	2	3	**1**	исходная последовательность
5	6	2	3	**1**	7	после 1-го прохода
5	2	3	**1**	6	7	после 2-го прохода
2	3	**1**	5	6	7	после 3-го прохода
2	**1**	3	5	6	7	после 4-го прохода
1	2	3	5	6	7	после 5-го прохода

В наихудшем случае потребуется $N-1$ проход по массиву. Однако если на некотором шаге последовательность уже стала упорядоченной (в частности, исходная последовательность может быть уже изначально упорядочена), то все последующие

проходы окажутся бесполезными. Уменьшить количество проходов можно, введя булевскую переменную b (флажок), которая указывает нам, есть ли необходимость в продолжении сортировки или последовательность уже упорядочена. Перед каждым проходом по массиву задаем $b = 0$. Если при очередном проходе для некоторой пары элементов отношение упорядоченности не выполняется, и мы эти элементы переставляем, мы меняем значение, меняем значение b на 1 (вскидываем флажок). Это значит, что последовательность еще не отсортирована, и нужно продолжать сортировку (выполнять еще один проход). Если после очередного прохода по массиву значение b осталось равно 0, то значит, что ни для одной пары соседних элементов отношение упорядоченности не нарушено. А это значит, по определению, что последовательность упорядочена.

Причины возможных ошибок Следует помнить, что для того, чтобы обменять значения двух переменных, следует воспользоваться третьей вспомогательной переменной.

1.21.2 Алгоритм сортировки методом нахождения локальных экстремумов

Иногда данный алгоритм "сортировки выбором". Существуют разновидности данного алгоритма:

- сортировка методом нахождения локального минимума;
- сортировка методом нахождения локального максимума;

Рассмотрим алгоритм сортировки методом нахождения локального минимума (алгоритм, использующий нахождение максимума – аналогичен).

Алгоритм: Проходим по массиву и находим минимум. Меняем местами начальный элемент массива и элемент массива, где находится минимум. Теперь минимальный элемент стоит на своем окончательном месте, его из рассмотрения исключаем и запускаем аналогичный процесс на подмассиве $a[1] \dots a[N-1]$. В наихудшем случае потребуется $N-1$ проход по массиву. Длина рассматриваемого подмассива на каждом шаге уменьшается.

7 5 6 2 3 **1** 4 первый проход, , $min = 1$

1	5	6	**2**	3	7	4	1 стоит на своем месте, второй проход, $min = 2$
1	*2*	6	5	**3**	7	4	третий проход
1	*2*	*3*	5	6	7	**4**	четвертый проход
1	*2*	*3*	*4*	6	7	**5**	пятый проход
1	*2*	*3*	*4*	*5*	7	**6**	шестой проход
1	*2*	*3*	*4*	*5*	*6*	*7*	окончательная последовательность

Причины возможных ошибок При нахождении очередного локального минимума необходимо знать не само значение минимального элемента, а индекс элемента подмассива, где находится этот минимальный элемент, чтобы впоследствии поменять местами начальный элемент подмассива и элемента, где находится минимум.

1.21.3 Алгоритм сортировки вставкой

Этот алгоритм основан на идее сохранения упорядоченности: пусть имеется упорядоченная последовательность. Требуется вставить в нее элемент таким образом, чтобы упорядоченность сохранилась.

Алгоритм: На каждом шаге алгоритма размер упорядоченного подмассива увеличивается с 1 до N. Подмассив длины 1, состоящий из элемента $a[0]$ упорядочен по определению. На каждом шаге j ($j = 1, \ldots, N-1$) элемент $a[j]$ вставляется в уже упорядоченную последовательность $a[0], \ldots a[j-1]$ следующим образом. Элемент $a[j]$ копируется во вспомогательную переменную c, тем самым освобождая j-ую ячейку массива. Упорядоченный подмассив $a[0], \ldots a[j-1]$ просматривается с конца. Все его элементы, большие элемента $c = a[j]$ (и которые, следовательно, должны стоять правее его), сдвигаются на разряд вправо. На освободившееся место записывается элемент c.

Пример: Упорядочить по возрастанию массив (5,3,2,4,1).

5	3	2	4	1	исходная последовательность
5					после 1-го прохода
3	5				после 2-го прохода
2	3	5			после 3-го прохода
2	3	4	5		после 4-го прохода
1	2	3	4	5	после 5-го прохода

1.21.4 Алгоритм сортировки фон-Неймана (сортировка слиянием)

Данный алгоритм основан на идее слияния двух упорядоченных последовательностей.

Алгоритм: Пусть, для простоты изложения длина сортируемой последовательности a равна $2*k$ для некоторого k. Для того, чтобы отсортировать последовательность a, делим ее пополам, сортируем отдельно каждую из половинок (длины 2^{k-1}), а затем сливаем две упорядоченные последовательности в одну. Для того, чтобы отсортировать последовательности длины 2^{k-1}, применяем тот же самый алгоритм, то есть делим каждую пополам, сортируем и сливаем. Продолжая рассуждать таким образом, мы дойдем, в конце концов, до последовательностей длины 1, которые по определению уже упорядочены. То есть весь алгоритм заключается в слиянии пар упорядоченных последовательностей сначала длины 1, затем 2, 4, 8 и так далее, пока не объединим две последовательности длины 2^{k-1} и не получим отсортированную последовательность длины $2 * k$. Каждую из подпоследовательностей в исходном массиве задаем индексами начала и конца сортируемой подпоследовательности.

Реализация данного алгоритма изящно реализуется рекурсией, которую мы будем рассматривать позже.

1.21.5 Сравнение алгоритмов сортировки

Достоинство первых трех изложенных выше алгоритмов сортировки – простота. Они не требуют дополнительной памяти, т.е. сортировка осуществляется на том же месте (несколько вспомогательных переменных, требуемых для обмена пар – не в счет). Однако временная сложность алгоритмов достаточно высокая: при сортировке массива длины N время работы ON^2. Для более эффективного алгоритма сортировки слиянием (а также других более быстрых алгоритмов сортировки, основанных на сравнении пар элементов), временная сложность $O(N \log N)$ (эта оценка является и нижней оценкой сложности). Недостаточная эффективность трех первых алгоритмов обуславливается следующим: любой алгоритм сортировки, в котором элементы всякий раз перемещаются на одну позицию, имеют сложность $O(N^2)$ (или больше). В алгоритме сортировки нахождением локальных экстремумов

на самом деле получается больше информации, чем непосредственно используется при перемещении соответствующего элемента в окончательную позицию, остальная информация "забывается". Другими словами, обмены элементов в первых трех изложенных алгоритмах имеют невысокий эффект. Однако при сортировке относительно небольших массивов рекомендуется применять именно эти алгоритмы, так как для таких массивов достоинства других алгоритмов в должной мере еще не проявляются.

Следует отметить, что алгоритм сортировки вставкой предпочтительно использовать для структур данных, для которых операция вставки выполняется эффективно (для структуры данных массив мы вынуждены многократно переписывать элементы массива, сдвигая их вправо, чтобы вставить элемент). Такой структурой данных являются, например, линейные списки, которые мы будем изучать позже.

1.22 Функции

Функции представляют собой важный инструмент, который позволяет писать хорошо структурированные программы. При этом в программах, написанных с использованием функций

- легче прослеживается основной алгоритм;
- программа проще в отладке;
- менее чувствительна к ошибкам программиста;
- программный код занимает меньше места, так как функция присутствует в программе в единственном экземпляре, а обращаться к ней можно многократно из разных точек программы.

При написании программы с использованием функций программа разбивается на отдельные подзадачи, каждая из которых решается по отдельности. В программах, написанных с использованием функций, реализуется идея нисходящего программирования.

Определение 1.14 Функция – *самостоятельный фрагмент программы, имеющий собственное имя и состоящий из группы операторов, решающих некоторую специфическую задачу. Функция связана с программой через параметры и выдает*

числовой результат, являющийся значением функции для данных входных параметров.

Собственно, любая программа на языке C++ – это последовательность функций, одна из которых имеет предопределенное имя *main*, с нее начинается выполнение программы. Все остальные функции получают управление, если производится их вызов. Исключение составляет функция *main*. Ее из программы на языке C++ вызвать нельзя. Вложенности функций в языке C++ нет.

1.22.1 Описание функции

Синтаксис описания функции:

type name (*список_ формальных_ параметров*)
{
  тело_ функции ;
}

где

- *type* – тип результата функции. Результат функции должен быть числовой. Если функция не возвращает никакого значения, то тип результата функции **void**;
- *name* – имя функции (произвольный идентификатор);
- *список_ формальных_ параметров* – список описаний формальных параметров, при помощи которых функция связывается с программой. Список формальных параметров имеет вид *type_ 1 param_ 1*, ..., *type_ k param_ k*, где *type_ i* и *param_ i* ($i = 1, \ldots, k$) соответственно тип и имя k-го формального параметра;
- *тело_ функции* – составной оператор (или блок), реализующий свойственные функции действия. В теле функции должен присутствовать оператор

  return (*выражение*);

где тип выражения должен совпадать с типом результата функции. Если тип результата функции **void** то оператор **return** может отсутствовать.

Описание функции без тела функции называется заголовком функции. Заголовок функции несет следующую нагрузку:

- указывает имя функции;
- список формальных параметров служит для связи функции с внешней средой (с функцией более высокого уровня). Через него мы указываем, что мы подаем на вход функции, и где и в каком виде будем получать результат.

Заголовок функции с последующей точкой с запятой называется *прототипом функции*. Прототип функции может присутствовать в программе несколько раз. Описание функции присутствует в программе только один раз.

1.22.2 Вызов функции

Вызов функции осуществляется при помощи оператора:

name (*список_ фактических_ параметров*)

Вызов функции может присутствовать в любом месте программы, где по синтаксису допустимо выражение того типа, какой формирует функция. Если тип функции не **void**, то вызов функции может входить в состав выражений.

При вызове функции фактические параметры подставляются на место соответствующих формальных параметров, и управление передается в начало функции. При этом должно выполняться соответствие фактических параметров формальным по

- типу,
- количеству,
- порядку.

1.22.3 Формальные и фактические параметры

- *Формальные параметры* – параметры, которые записываются в заголовке после имени функции при ее описании;
- *Фактические параметры* – параметры, которые подставляются вместо формальных параметров при вызове функции.

Параметры, используемые при описании функции, называются формальными, потому, что они не являются переменными в программистском смысле. Они не служат именами ячеек памяти и не содержат никаких конкретных значений. Блок операторов, составляющий тело функции, не производит никаких действий над этими параметрами, а лишь задает характер, шаблон действий, которые будут производиться на самом деле при вызове функции.

1.22.4 Локальные и глобальные переменные

Одним из атрибутов переменной является область ее действия.

- *Локальные переменные* функции – переменные, описанные внутри функции. Область действия этих переменных – это блок, внутри которого они описаны. Данные переменные недоступны в других функциях. Локальные переменные в разных функциях могут иметь одинаковые имена – это разные переменные с разными областями действия.
- *Глобальные переменные* – это переменные, описанные вне любой функции. Они доступны внутри любой функции. Область действия таких переменных – вся программа.

Операторы в описании тела функции могут оперировать локальными переменными, глобальными переменными и формальными параметрами. Попытка обратиться в локальной переменной другой функции будет ошибкой. При объявлении локальной переменной функции с именем, совпадающим с именем глобальной переменной, данная локальная переменная перекрывает глобальную. Для обращения к перекрытой глобальной переменной можно использовать операцию "::" – "операцию разрешения области видимости".

Использование глобальных переменных в функции – это еще один способ возврата результата работы функции. Часто его называют "побочным эффектом". Такое

название следует из не очевидности механизма работы функции, так как из вызова функции не видно, что будут устанавливаться значения глобальных переменных. Несмотря на простоту использования такого способа возврата значения из функции, не следует злоупотреблять этим способом, так как это затрудняет понимание работы функции и ее модификацию.

1.22.5 Передача параметров по значению и по ссылке

Существуют два способа передачи параметров в функцию: по значению и по ссылке.

- параметр, передаваемый в функцию *по значению* оформляется в списке формальных параметров следующим образом:

  *type_ param_*

  При вызове функции в качестве соответствующего фактического параметра могут быть использованы:

  - фактическая константа (имеющая тип, совпадающий с типом соответствующего формального параметра);
  - имя переменной (имеющей тип, совпадающий с типом соответствующего формального параметра);
  - выражение (имеющее тип, совпадающий с типом соответствующего формального параметра).

  При вызове функции вместо соответствующего формального параметра в функцию будет передаваться *значение* соответствующего фактического параметра, которое будет скопировано в локальную переменную функции, соответствующую данному формальному параметру. Соответственно, все изменения, которые функция произведет над данным параметром, никак не отразятся на состоянии переменной, которая была использована в качестве соответствующего фактического параметра.

- параметр, передаваемый в функция *по ссылке* оформляется в списке формальных параметров следующим образом:

*type_* &*param_*

При вызове функции в качестве соответствующего фактического параметра может быть использовано только имя переменной;

При вызове функции вместо соответствующего формального параметра в функцию будет передаваться *сама переменная*, задающая фактический параметр (а именно ее адрес). Все изменения, которые функция произведет над данным параметром, будут производиться именно с этой переменной и после выхода из функции значение этой переменной будет измененным.

При передаче массива в качестве параметра в функцию он может быть передан только по ссылке (но знак & для него писать не надо).

1.22.6 Входные и выходные параметры

Функция связывается с внешней средой (с функцией, откуда был произведен ее вызов) через параметры. При этом параметры функции можно разделить на *входные* и *выходные*.

- Входные параметры задают те значения, которые должны быть поданы на вход функции для того, чтобы она смогла решить поставленную задачу.
- Выходные параметры определяют имена переменных, через которые функция будет возвращать результат работы во внешнюю среду.

Входные параметры могут передаваться в функцию по значению и по ссылке. Передачу по ссылке для входных параметров используют в тех случаях, когда хотят сэкономить время на пересылку фактического параметра в локальную память функции, если переменная занимает большой объем памяти. Для того, чтобы защитить входной параметр от нежелательного изменения, используют спецификатор **const** перед соответствующим формальным параметром.

Выходные параметры должны передаваться в функцию только то ссылке. Выходной параметр указывает функции имя переменной, где она должна разместить результат. Это еще один способ возврата результата функции, кроме возврата оператором **return**, и может использоваться, если функция вырабатывает более чем один результат или результат работы не является числовым.

1.22.7 Параметры по умолчанию

Для того, чтобы упростить вызов функции, для формальных входных параметров функции можно указать значения, используемые по умолчанию. Данные параметры должны идти последними в списке параметров и могут опускаться при вызове функции. В случае их отсутствие при вызове функции в качестве значений, подставляемых вместо соответствующих формальных параметров, используются значения по умолчанию. В качестве значений по умолчанию могут использоваться константы, глобальные переменные и выражения.

1.22.8 Модифицированное тело функции

Семантика обращения к функции очевидна – это вычислений функции для данного набора фактических параметров. Модифицированным (измененным) телом функции называется обозначение для записи алгоритма без использования функции. При этом выполняется следующее:

- для каждой локальной переменной функции создается новая переменная, не использованная в программе;
- имена формальных параметров, передаваемых по ссылке, заменяются на имена переменных, используемых в качестве соответствующих фактических параметров;
- для каждого формального параметра, передаваемого в функцию по значению, создается новая переменная, не использованная в программе. Значение соответствующего фактического параметра вычисляется и используется в качестве начального значения данной переменной.

1.22.9 Перегрузка функций

В языке C++ можно определять несколько разных функций с одним и тем же именем. При этом разными считаются функции, которые отличаются друг от друга количеством или типами формальных параметров. Отличие только по типу результата функции не допускается.

В этом случае говорят о перегруженных функциях.

Определение 1.15 Перегруженные функции – *функции, имеющие одинаковое имя, но разное количество или типы формальных параметров.*

При вызове функции компилятор по имени функции определяет, является ли она перегруженной, и если она перегружена, то по типу и количеству определяет, какую именно функцию необходимо вызвать.

1.22.10 inline-функции

Функции, определяемые со спецификатором **inline** перед заголовком функции, называются *встроенными* функциями. Для таких функций их вызов производится не обычным для функций образом, через стек, а простой подстановкой модифицированного тела функции в то место программы, откуда производится ее вызов. Такой механизм может использоваться для коротких функций, для которых накладные расходы по обычному вызову функции на фоне небольшого ее кода становятся неоправданными.

1.23 Структуры

Структурный тип является производным типом данных. Мы уже знакомы с производным типом данных таким, как массив. Переменная типа массив хранит под одним именем множество различных значений. При этом все элементы массива должны быть одного и того же типа. Однако довольно часто бывает необходимо собрать под одним именем группу разнородных данных. Например, информация о студенте факультета может включать в себя следующие данные: фамилия, имя, год рождения, пол, семейное положение, средний бал за сессию и т.д. Понятно, если мы хотим хранить данные на одного студента в одной переменной, то эта переменная должна иметь такой тип, который бы позволял ей объединять разнотипную информацию.

Таким типом является структурный тип. Переменные данного типа называют структурами. Таким образом, структура – это есть набор данных разного типа (полей структуры). Разные структурные типы отличаются друг от друга именами, составом и порядком следования своих полей. Чтобы можно было определять переменные конкретного структурного типа, этот тип должен иметь имя. То есть

сначала определяют структурные тип, а потом описывают структуры данного типа. Можно определить и неименованные структуры (не имеющие имени типа).

Дадим определение синтаксиса и семантики структурного типа:

Синтаксис:

struct structName
{
$type_1\, name_1$;
...
$type_k\, name_k$;
};

где

- structName – имя структурного типа;
- $name_1, \ldots, name_k$ – имена полей структурного типа, имеющих типы
- $type_1, \ldots, type_k$, соответственно
- (structName, $name_1, \ldots, name_k$ – произвольные идентификаторы).
- имена полей $name_1, \ldots, name_k$ должны быть уникальными в пределах одного структурного типа, но могут совпадать с именами полей в другом структурном типе, а также с именами других переменных. Типы полей могут быть любыми, кроме типа этой же структуры.

Семантика: Определяется структурный тип с именем structName. Любая переменная типа structName (структура данного структурного типа) будет иметь поля $name_1, \ldots, name_k$, имеющие типы $type_1, \ldots, type_k$, соответственно.

Когда определен структурный тип, можно определять переменные этого типа:

structName var1, var2, var3;

Каждая переменная var1, var2, var3 имеет поля $name_1, \ldots, name_k$.

Существует другой способ определить переменную типа структура – без объявления структурного типа:

struct

{

$type_1\ name_1$;

...

$type_k\ name_k$;

} var1, var2, var3

Однако предпочтительно использовать первый способ, поскольку он делает программу более наглядной. Кроме того, без имени типа в некоторых случаях не обойтись, например, при использовании структуры в качестве параметра функции.

Пример:

```
struct tstruct
{
  int x;
  double y,z;
  bool c;
};
tstruct s1,s2,s3;
```

1.23.1 Операции над структурами.

- **Доступ к полям структуры** осуществляется через операцию “точка”.

  Например, s1.x, s2.c.

- Разрешено присваивание одной структуры другой. При этом копируются все поля структуры.

  То есть операция $s1 = s2$; означает

  s1.x=s2.x;

  s1.y=s2.y;

  s1.z=s2.z;

  s1.c=s2.c;

  Как следствие, разрешено передавать структуру в функцию по значению.

- Нет групповых операций над структурами. Все операции необходимо проводить над полями структуры.

Причины возможных ошибок при работе со структурами:

1. После закрывающейся фигурной скобки в определении структурного типа необходима точка с запятой;

2. При присваивании одной структуры происходит копирование всех полей. Однако если полем структуры является массив, необходимо производить копирование элементов массива, поскольку при присвоении одного массива другому не происходит копирование его элементов.

3. Одной из причин ошибок является попытка использования групповых операций над структурами.

1.23.2 Сравнение структур и массивов.

В таблице ниже приведен сравнительный анализ структурного типа данных и типа данных массив.

массивы	**структуры**
фиксированный размер	
прямой доступ к элементам:	
при помощи вычислимого индекса	при помощи имени
нет групповых операций	
не разрешена операция присваивания	разрешена операция присваивания
в функцию может передаваться:	
только по ссылке	по ссылке или по значению

1.23.3 Бинарные файлы из структур

При работе с бинарными файлами, элементами которого являются структуры, используется неформатированный ввод/вывод. При этом операция чтения из файла должна считывать в преременную структурного типа целиком структуру и лишь затем с использованием операции “точка” производится обращение к тем или иным полям структуры. Аналогично, при выводе информации в файл необходимо подготовить выводную структуру, заполнив в ней все поля и только затем командой

вывода данная структура отправляется в файл. При попытке считывать из файла отдельно каждое поле структуры, адресуясь к нему напрямую, возможна неправильная работа программы. Это связано с тем, что для увеличения быстродействия программы компилятор выделяет не сплошной и непрерывный блок памяти под размещение полей переменной структурного типа, а блок памяти, в котором имеются внутренние пустоты. Эти пустоты образуются потому, что компилятор располагает целые и вещественные поля структуры с адресов, кратным некоторому числу.

Пример: Дан бинарный файл из структур, содержащий информацию о студентах факультета ВМК: фамилия студента, номер группы, оценки за последнюю сессию (пять оценок за экзамены и три отметки о сданных зачетах). Сформировать бинарный файл из структур, содержащий информацию о студентах-отличниках. Программа приведена на рис. 1.10.

1.24 Рекурсивные функции

Определение 1.16 Рекурсивная функция – *это функция, которая прямо или косвенно вызывает сама себя.*

Пример:

Прямая рекурсия:

```
void f()
{
  ...
  f();
  ...
}
```

Косвенная рекурсия:

```
void B();
void A()
```

```
#include <iostream.h>
#include <fstream.h>
#include <string.h>
void main()
{
 struct tinfo         //тип структуры
 {  char fam[50];     //для исходного файла
    int gruppa;
    int marks[5];
    bool zach[3];
  };
 struct totl          //тип структуры
 {  char fam[50];     //для результирующего файла
    int gruppa;
 };
 tinfo info;
 totl otl;
 ifstream myin("stud.bin", ios::binary);
 ofstream myout("otlich.bin", ios::binary);
 myin.read((char*) &info, sizeof(tinfo));  //вводим первого студента
 while (!myin.eof())
 {
  bool b=true;                             //проверяем, отличник ли он
  for (int i=0; i<5; i++)
      if (info.marks[i]!=5) {b=false; break;}
  for (int i=0; i<3; i++)
      if (info.zach[i]!=true) {b=false; break;}
  if (b)                                   //если он отличник
  {
   strcpy(otl.fam,info.fam);               //подготавливаем структуру
   otl.gruppa=info.gruppa;                 //для вывода
   myout.write((char*) &otl, sizeof(totl));  // выводим структуру
  }
  myin.read((char*) &info, sizeof(tinfo));   //вводим нового студента
 }
 myin.close();
 myout.close();
}
```

Рис. 1.10: Работа с бинарными файлами из структур.

```
{
  ...
  B();
  ...
}
void B()
{
  ...
  A();
  ...
}
```

Рекурсивные функции чрезвычайно полезны. Их используют для компактной реализации рекурсивных алгоритмов, а также для работы с рекурсивно определенными структурами данных.

Для построения рекурсивного алгоритма для некоторой задачи необходимо свести задачу к самой же себе, но меньшей размерности. Например, рассмотрим задачу вычисления факториала.

Итак, для заданного натурального числа n вычислить $y = n!$. Естественный знакомый нам итеративный алгоритм решения данной задачи использует домножение переменной y последовательно на все числа от 2 до n. Ниже приведена функция, соответствующая данному алгоритму.

```
int Factor(int n)
{
  int y;
  y=1;
  for (int i=2; i<n; i++)
  {
    y=y * i;
  }
  return y;
}
```

Для построения рекурсивного алгоритма для данной задачи сведем ее к самой себе, понизив размерность задачи:

$y_n = y_{n-1} * n$, при $n > 1$;

$y_n = 1$, если $n = 1$

То есть, если мы вычислим факториал числа $n - 1$, останется домножить его на n, чтобы получить $n!$. Как же вычислить $(n - 1)!$? Напишем функцию, которая вычисляет факториал для произвольного n, и будем использовать ее для вычисления $(n - 1)!$ Соответственно этому алгоритму имеем следующую (рекурсивную) функцию для вычисления факториала:

```
int Factor(int n)
{
  if (n==1)
  return 1;
  else return (n * Factor(n-1));
}
```

Разберем основные моменты данной рекурсивной функций:

1. Рекурсивный вызов производится, если условие $(n == 1)$ не выполняется, в противном случае рекурсия останавливается. Это чрезвычайно важный момент, так как если функция будет только из оператора $return(n*Factor(n-1))$, то мы будем иметь так называемую "бесконечную рекурсию", которая никогда не останавливается. В этом случае функция будет снова и снова вызывать сама себя, понижая значение параметра как $n, n-1, \ldots, 1, 0, -1, \ldots$ – программа зациклится. Но такое зацикливание в отличие от бесконечных циклов потенциально гораздо более опасно. Как мы знаем, при каждом вызове функции в стеке выделяется новый кусок памяти, в результате стек неограниченно растет, а объем его фиксирован. В конце концов, произойдет переполнение стека, и программа аварийно завершится.

2. Та же ситуация произойдет, если мы поменяем ветки then и else, сменив условие на $(n! = 1)$. В этом случае выполнение функции до ветки else так и не дойдет, так как будет бесконечно вызывать сама себя.

3. Обратим внимание, что в выражение $(n * Factor(n-1))$ стоит вызов функции, где вместо формального параметра подставлен фактический. Не следует вызов функции путать с описанием функции.

Нисходящая и восходящая ветви рекурсии. Разберем, как происходит выполнение рекурсивной функции на том же примере вычисления факториала. Пусть нам надо вычислить 4!. В программе вызываем нашу рекурсивную функцию для $n = 4$.

```
int y=Factor(4);
```

Согласно алгоритму, в теле функции Factor опять происходит вызов той же самой функции но для параметра $n = 3$. То есть функция Factor для параметра $n = 4$ не может возвратить результат, пока не получит результат от Factor(3). Далее Factor(3) вызывает Factor(2), в теле Factor(2) происходит вызов Factor(1). При выполнении функции Factor(1) рекурсия останавливается, поскольку в данном случае $n == 1$. Говорят, что закончилась *"нисходящая ветвь рекурсивного алгоритма"*.

Далее начинается *"восходящая ветвь рекурсивного алгоритма"*. Factor(1) возвращает в качестве результата значение 1 и управление передается в функцию Factor(2), откуда Factor(1) была вызвана. Factor(2) возвращает значение 2*1 в функцию Factor(3) и т.д. до тех пор, пока оператор return из функции Factor(4) не осуществит возврат результата 4*2*3*1 внешней функции, откуда был произведен самый первый вызов.

Оправданность применения рекурсии. Рекурсивные функции, как правило, записываются более компактно, код занимает меньше места и выглядит красиво. Но следует хорошо подумать, прежде чем отдать предпочтение рекурсивному алгоритму, поскольку не всегда то, что кратко записано, так же кратко и исполняется. Приведем пример.

Пример: Вычисление n-го числа Фибоначчи.

Дано натуральное n. Найти n-е число Фибоначчи, которое определяется рекуррентной формулой

$\Phi_n = \Phi_{n-1} + \Phi_{n-2}$, $\Phi_0 = 0, \Phi_1 = 1$.

Как мы видим, сама формула для определения значения чисел Фибоначчи содержит рекурсию. И нетрудно написать соответствующую рекурсивную функцию

```
int Fib(int n)
{
  if (n==0) return 0;
  if (n==1) return 1;
  return ( Fib(n-1)+Fib(n-2));
}
```

Однако посмотрит, как будет происходить выполнение данной функции. Пусть нужно найти значение числа Фибоначчи для $n = 5$.

Можно построить дерево нисходящего процесса рекурсии. Итак, функция Fib(5) производит два вызова:
Fib(4) и Fib(3).
Fib(4) вызывает Fib(3) и Fib(2).
Fib(3) вызывает Fib(2) и Fib(1).
Fib(3) в теле функции Fib(4) опять вызывает Fib(2) и Fib(1).
и т.д. То есть мы видим, что происходит повторный выход функции Fib для одинаковых значений параметра n, что приводит к неэффективно долгому выполнению данной задачи.

При написании обычного рекуррентного алгоритма решения данной задачи такой проблемы не возникает, и в данном случае применение рекурсии совершенно не оправдано.

1.25 Ссылочный тип данных

Ссылка – это синоним имени переменной, указанной при инициализации ссылки. Ее можно рассматривать как указатель, который всегда автоматически разименовывается.

При этом должно выполняться следующее:

- ссылка должна явно инициализироваться при ее описании;

- после инициализации ссылке не может быть присвоена другая переменная. Ссылка жестко закрепляется за переменной, на которую она настраивается при описании;
- тип ссылки совпадает с типом величины, на которую она ссылается;
- нельзя определять указатели на ссылки, массивы ссылок и ссылки на ссылки.

Ссылка не занимает дополнительного места в памяти и является просто другим именем величины. Операция над ссылкой приводит к операции над величиной, на которую она ссылается.

Например:

int x=4;	объявляется переменная x типа **int**, ей присваивается значение 4
int& px=x;	переменная px – ссылка на переменную x (ее псевдоним, другое имя) изменяя значение px мы изменяем значение x и наоборот
px=5;	значение x стало равно 5
x=6;	значение px стало равно 6

1.26 Тип указатель

Переменные, с которыми может работать программа, делятся на статические и динамические.

Статические переменные – это переменные, память под которые выделяется на этапе трансляции. Все переменные, с которыми мы имели дело до сих пор – статические переменные. Когда в программе встречается объявление переменной, компилятор выделяет под эту переменную участок памяти размером в соответствии с типом этой переменной и связывает имя переменной с этим участком памяти. Все обращения к данной переменной по ее имени в программе заменяются на обращения к участку памяти, соответствующем данной переменной.

Динамические переменные – это переменные, память под которые выделяется программистом на этапе выполнения программы с помощью средств языка. Динамические переменные используются в программах в ситуациях, когда программист не знает заранее, сколько памяти может потребоваться и поэтому не может заранее зарезервировать необходимый участок статическим образом. Также использование

динамических переменных предоставляет механизм для создания гибких структур данных, таких, как списки, деревья, позволяющих эффективно обрабатывать информацию.

Для работы с динамическими переменными в языке C++ используется тип данных "указатель" (англ. pointer). Переменная типа указатель – это статическая переменная (память под нее выделяется на этапе трансляции), но значение, которое может храниться в данной переменной, указывает на динамическую переменную. А именно, в переменной типа указатель хранится адрес области памяти, выделенной программистом под динамическую переменную.

Синтаксис описания переменной типа указатель:

type **name*;

где *name* – имя переменной типа указатель (произвольный идентификатор),
type – тип динамической переменной, на которую может указывать указатель *name*.

Семантика описания переменной типа указатель:

Создается статическая переменная *name* типа указатель. В данной переменной мы можем хранить адрес участка памяти, где может располагаться динамическая переменная типа *type*. Пока в переменной *name* нет никакого конкретного значения (она не инициализирована). Самой динамической переменной типа *type* пока нет, память под нее еще не выделена.

1.26.1 Выделение и освобождения памяти под динамические переменные

Для того чтобы выделить память под динамическую переменную, используется команда **new**.

Синтаксис команды **new**:

name = new *type*;

Семантика команды **new**: Ищется участок свободной памяти, достаточной для размещения переменной типа *type*. Этот участок помечается как занятый, адрес этого участка памяти помещается в переменную *name*. Область памяти, из которой происходит резервирование памяти под динамические переменные, называется динамической памятью, или кучей (англ. heap).

Выделенная командой **new** память считается занятой с момента выполнения команды **new** до конца программы. Для того чтобы иметь возможность повторного использования памяти для других динамических переменных, память под ненужные больше динамические переменные необходимо освобождать (помечать как свободную). Это делается с помощью команды освобождения памяти **delete**, которую можно считать обратной к команде выделения памяти.

Синтаксис команды **delete**:

delete *name*;

Семантика команды **delete**: Участок памяти, адрес которой хранится в переменной *name*, помечается как свободный. Этот участок, возможно, использовать для дальнейшего резервирования.

1.26.2 Обращение к динамическим переменным

Таким образом, инициализировать переменную типа указатель возможно, выделив командой **new** память под динамическую переменную. При выделении участка памяти его адрес автоматически помещается в переменную типа указатель. Зная адрес динамической переменной, можно к ней обратиться. Выполняется это с помощью операции разадресации "*".

Если *name* – указатель на динамическую переменную, то **name* – сама динамическая переменная.

Например:

int *p; объявляется переменная типа указатель p
p=new **int**; выделяется память под переменную типа **int**,

	адрес этой памяти засылается в p
*p=4	динамической переменной, адрес которой находится в p,
	присваивается значение 4

1.26.3 Пустой указатель

Как мы говорили, когда переменная типа указатель описана до выполнения команды new с этой переменной, в ней не хранится никакого конкретного значения. Чтобы различать ситуации, когда переменная не определена и когда она "ни на что не ссылается", т.е. не содержит адрес конкретного участка памяти, введено понятие пустого указателя.

Значение пустого указателя хранится в константе NULL и равно нулю. Чтобы использовать данную константу в программе необходимо подключить следующий файл заголовков:

#include<stdlib.h>

1.26.4 Указатель на статические переменные

Переменная типа указатель может ссылаться и на статические переменные. Для этого при инициализации переменной типа указатель нужно использовать операцию взятия адреса "&":

int *p;	объявляется статическая переменная типа указатель p
int x=4;	объявляется статическая переменная типа **int** x,
	которой присваивается значение 4
p=&x;	указатель p указывает на статическую переменную x

1.26.5 Операции над указателями

Кроме операции разадресации с указателями можно выполнять следующие операции: сложение с константой, вычитание, инкремент, декремент. Операции можно выполнять только над указателями одинаковых типов. Данные операции имеют

смысл в основном при работе со структурами данных, последовательно размещенными в памяти, например, с массивами. При этом автоматически учитывается размер типа переменных, на которые указывают указатели. То есть, например, разность двух указателей – это разность их значений, деленная на размер типа элементов, на которые они указывают. Инкремент перемещает указатель к следующему элементу, сдвигаясь на такое количество байтов, которое занимает тип, на который ссылается данный указатель.

1.26.6 Динамические массивы

С помощью команды new можно создавать динамические массивы, выделяя под них столько памяти, сколько требуется.

Синтаксис команды создания динамического массива.

type **name*;
name=**new** *type*[*size*];

где *type*– тип элементов массива, *name* – имя создаваемого массива, *size* – размер массива (количество элементов).

Семантика команды создания динамического массива.

Выделяется непрерывный участок памяти размера *size**sizeof(*type*). Адрес этого участка памяти заносится в переменную *name*.

Освобождение памяти для динамического массива выполняется с помощью команды *delete*.

Синтаксис команды освобождения динамического массива.

delete []*name*;

Семантика команды освобождения динамического массива.

Участок памяти по адресу *name* помечается как свободный.

Операции с динамическим массивом можно осуществлять как в синтаксисе указателей, так и с помощью индексов.

Глава 2

Динамические структуры данных

2.1 Списки

При разработке любого алгоритма решающее значение имеет выбор структуры данных для представления обрабатываемой информации. От выбора такой структуры данных зависит алгоритм решения задачи. Например, использование такого типа данных, как массив, предполагает возможность быстрого прямого индексированного доступа к обрабатываемым элементам. Однако у такой структуры данных существуют и свои недостатки. Прежде всего, размер массива должен быть определен до начала выполнения программы (так как память под статический массив выделяется на этапе компиляции). Однако довольно часто бывает неизвестно, сколько элементов придется обрабатывать и, следовательно, сколько памяти может потребоваться под массив. Использование для хранения обрабатываемой информации файлов позволяет хранить последовательности элементов потенциально неограниченной длины. Однако и для массивов и для файлов неэффективно выполняются такие операции, как вставка элемента, (поскольку память под массив выделяется сплошным куском). Допустим, имеем текст

Слова, слова, . . . , слова

и требуется вставить в середину текста новые слова, чтобы получилось

Слова, слова, . . . , новые слова, . . . , слова

Если мы работаем со структурой данных, где элементы расположены последовательно друг за другом (массив, файл), то нам не остается ничего другого, как

переписать сначала первую часть текста, потом – новые слова, и оставшуюся часть слов.

Использование различных динамических структур данных позволяет организовать хранение обрабатываемой информации таким образом, чтобы:

- размер такой структуры (количество элементов) был потенциально неограниченным;
- и можно было эффективно модифицировать такую структуру данных, выполняя операции вставки, замены, удаления элементов.

Одной из таких структур данных являются линейные односвязные списки (а также различные их разновидности, такие, как стек, очередь, циклические списки и т.д.). Такую структуру данных определяет программист, то есть ее можно назвать пользовательской структурой данных. Реализуется такая структура при помощи динамических переменных, поэтому ее называют динамической структурой данных.

Для организации таких динамических структур данных используют прием, когда каждый элемент хранит адрес (указатель) некоторого другого элемента структуры данных.

Линейный односвязный список – это такая структура данных для представления последовательности элементов, в которой каждый элемент состоит из информационной части (где собственно, хранится информационное значение элемента), и указателя на следующий элемент последовательности. При динамической реализации линейного односвязного списка тип каждого его элемента может быть задан в виде структуры

```
struct elem
{
  tinfo info;
  elem * next;
};
```

где *info* – информационное поле элемента списка (то есть, информация, которую мы и хотим хранить в такой структуре данных), *next* – указатель на следующий элемент списка. Тип информационного поля *tinfo* – это тип информации, хранимой

в списке. Например, для работы со списком целых чисел типом поля *info* будет являться **int**.

Чтобы задать список, достаточно задать указатель на самый первый элемент списка. Будем хранить указатель на начало списка в переменной с именем *top*. Пройдя по указателям *next*, можно осуществить доступ к любому элементу списка. Последний элемент списка отличается од других элементов тем, что не имеет следующего за ним, поэтому в поле *next* этого элемента будем хранить значение пустой ссылки NULL.

Поскольку в такой структуре данных из каждого элемента имеется доступ к следующему элементы списка, но не имеется доступа к предыдущему, то при работе со списком желательно не менять значение указателя *top* на голову списка, поскольку, если мы сдвинем этот указатель на следующий, назад вернуться мы уже не сможем. Для доступа к элементам списка используют текущий указатель p.

elem *top, *p;

Поскольку p – указатель на текущий элемент списка, то для того, чтобы получить доступ к информационному значению текущего элемента, необходимо выполнить операцию разадресации (получив доступ к самой структуре – текущему элементу списка), а затем взять поле *info* структуры, т.е. *(p*).info*. Скобки используются, поскольку операция доступа к полям структуры '.' имеет более высокий приоритет, чем операция разадресации '*'. Для такой операции (доступа к полям структуры через указатель на структуру) используют более краткую и более удобную запись *p->info*. Аналогично, запись *p->next* обозначает доступ к полю *next* элемента списка, на который указывает указатель p.

2.1.1 Основные операции для работы с линейным односвязным списком

Основные операции над списками разделим на два типа: для которых есть аналог в последовательных типах данных, и для которых такого аналога нет. К первому типу операции отнесем следующие операции: установка на начало последовательности, сдвиг к следующему элементу, проверка на окончание последовательности, извлечение и запись значения текущего элемента. Для сравнения, данные опера-

ции приведем для случаев, если последовательность хранится в массиве (доступ к текущему элементу производится при помощи индекса) и если последовательность хранится в линейном односвязном списке (доступ к текущему элементу производится при помощи указателя).

Пусть последовательность обрабатываемых элементов представлена в виде списка, заданного указателем на начало *top*, в первом случае, и массивом a длины N, во втором случае.

список	**массив**
установка на начало последовательности	
p=top;	i=0;
сдвиг к следующему элементу последовательности	
p=p->next;	i=i+1;
извлечение текущего элемента последовательности	
x=p->info;	x=a[i];
запись текущего элемента последовательности	
p->info=x;	a[i]=x;
проверка на конец последовательности	
(p!=NULL)	(i!=N)

Ко второй группе рассматриваемых нами операций относятся операции, для которых не имеется аналога в таких последовательных структурах данных, как массив. К таким операциям отнесем вставку элемента и удаление элемента в список. Пусть список задан указателем *top* на начало списка. Переменная p – указатель на текущий элемент списка.

```
elem *top, *p;
```

Вставка (добавление) элемента в список.

- Вставка элемента в начало списка. В начало списка вставляется новый элемент, указатель *top* устанавливается на добавленный элемент.

```
elem *q=new elem;        создаем новый элемент
```

q->next=top;	прицепляем его перед началом списка
top=q;	начало списка меняется

- Вставка элемента в список после текущего указателя p.

elem *q=p->next;	сохраняем в q адрес элемента, следующего после p
p->next=new elem;	создаем новый элемент, прицепляем его за элементом с указателем p
p->next=new elem;	связываем его с элементом q

Удаление элемента из списка.

- Удаление первого элемента списка. Первый элемент списка удаляется, начальный указатель устанавливается на второй элемент (фактическое начало списка после удаления первого элемента).

elem *q=top;	сохраняем указатель на первый элемент,
	чтобы иметь возможность его удалить
top=top->next;	список теперь будет начинаться со второго элемента
delete q;	удаляем первый элемент

- удаление элемента после текущего элемента (после элемента с указателем p).

elem *q=p->next;	сохраняем в q адрес элемента, следующего после p
p->next=q->next;	перекидываем ссылку через один элемент
delete q;	удаляем элемент

2.1.2 Статическая реализация линейных списков

Если количество потенциальных элементов списка небольшое и заранее известно, то списки можно реализовывать при помощи массивов. Каждый элемент массива может в этом случае содержать информационное поле и индекс элемента массива, являющегося следующим элементом списка. Начало списка задается индексом *top*

элемента массива, являющегося первым элементом списка. Текущий элемент задается индексом p. Операции над списком в этом случае реализуются через операции над индексами.

```
struct elem
{
  tinfo info;
  int next;
}
const int N=100;
elem spisok[N];
int top, p;
```

2.1.3 Другие варианты линейных списков

Можно также рассматривать другие различные варианты линейных списков:

- циклический список: после последнего элемента списка идет первый элемент списка (после *next* последнего элемента содержит не NULL, а адрес первого элемента, т.е. *p->next=top;*)
- двусвязный линейный список: каждый элемент содержит адрес следующего элемента и адрес предыдущего элемента списка. Таким образом, существует возможность сдвига по списку в любом направлении, как вперед, так и назад.

2.2 Рекурсивное определение списков

Линейные односвязные списки можно определять рекурсивным образом. Основываясь на данном рекурсивном определении, можно определять рекурсивные функции для работы со списками.

Определение 2.1

- *По определению пустой список T – это список.*
- *Если список T не пуст, то в нем существует элемент начало (или голова) списка, и хвост списка, следующий за начальным элементом и являющийся, в свою очередь, тоже списком;*

Основываясь на данном рекурсивном определении списка, динамическую реализацию списка можно определить рекурсивным образом. Переменная *top* задает список и является указателем на его начало:

- если список пуст, то *top* – это пустой указатель: $top = NULL$;
- если список не пуст, то в нем существует, по крайней мере, один элемент – начало списка, переменная *top* хранит указатель на этот элемент;
- сам элемент – голова списка представлен в виде структуры из двух полей
  - информационное поле *info*;
  - указатель *next* на список, являющийся хвостом исходного списка.

Основываясь на данном рекурсивном определении, можно определять рекурсивные функции для работы со списком. Как правило, для обработки списка необходимо обработать голову списка и оставшуюся часть списка – хвост списка. Для обработки хвоста можно использовать рекурсивный вызов функции.

Пример: На рис. 2.1 приведена рекурсивная функция для нахождения суммы элементов списка из целых чисел, основанная на следующем рекурсивном алгоритме: для вычисления суммы элементов списка необходимо найти сумму элементов списка, являющегося хвостом исходного списка и прибавить к результату значение, хранящееся в голове исходного списка.

Пример: На рис. 2.2 приведена рекурсивная функция для построения копии списка.

2.3 Стек

Определение 2.2 Стек – *это структура данных, организованная по принципу "последний пришел – первый ушел" (в англоязычной литературе используется термин LIFO – Last In First Out).*

Как правило, структура данных стек предназначена для временного хранения элементов с целью их последующей обработки. Можно считать, что стек – это последовательность элементов (возможно пустая), в которой все включения и исклю-

```
#include <stdlib.h>
struct elem
{
 int info;
 elem *next;
};

int sum(elem *top)
{
 if (top==NULL)
    return 0;
 return (top->info+sum(top->next));
}

int main(int argc, char* argv[])
{
        return 0;
}
```

Рис. 2.1: Рекурсивная функция нахождения суммы элементов списка

```
#include <stdlib.h>
struct elem
{
 int info;
 elem *next;
};

void copy(elem *top1, elem* &top2)
{
 if (top1==NULL)
    top2=NULL;
 else
 {
  top2=new elem;
  top2->info=top1->info;
  copy(top1->next, top2->next);
 }
}

int main(int argc, char* argv[])
{
        return 0;
}
```

Рис. 2.2: Рекурсивная функция построения копии списка

чения элементов производят с одного конца последовательности, называемого вершиной, или головой стека. Количество элементов, находящихся в стеке называют *глубиной* стека.

Чтобы нагляднее представить себе принцип устройства стека, можно вообразить узкую пробирку с таблетками внутри. Мы можем без труда извлечь из пробирки таблетку, находящуюся в самом верху пробирки. И для того, чтобы извлечь таблетку, помещенную в пробирку самой первой, необходимо полностью опустошить пробирку. В нашей повседневной жизни существует множество примеров, устроенных по принципу стека: магазин с патронами, коробка с бумагой для принтера, и т.д. В программировании также широко используется принцип стека. Напомним, что по принципу стека выделяется память под локальные переменные функций. Стеки также используются в системном программном обеспечении, в рекурсивных алгоритмах.

Пусть S – стек (реализованный некоторым образом), *tinfo* – тип элементов стека. Для того чтобы работать со стеком, необходимо уметь выполнять следующие основные операции:

1. **void** Create(Stek &S); – инициализация стека (создается пустой стек, в нем пока нет ни одного элемента);
2. **void** Push(Stek &S, tinfo x); – положить элемент x в стек;
3. tinfo Push(Stek &S); извлечь элемент из стека (элемент удаляется из стека, значение удаленного элемента – результат данной операции);
4. **bool** Empty(Stek S); – проверка стека на пустоту.

Отметим, что в первых трех операциях состояние стека меняется, поэтому важно передавать параметр стек в функцию по ссылке.

В программе стек можно реализовывать при помощи различных типов данных.

Статическая реализация стека. Если значение N максимальной глубины стека невелико и заранее известно, то стек можно реализовать при помощи массива, тем самым, выделяя для него непрерывную область памяти. Преимущество такой реализации – малые затраты по памяти и малое время реализации операций над стеком.

```
const int N=100;
struct stek
{
   tinfo info[N];
   int top;
};
stek S;
```

Поле *top* содержит индекс первой свободной ячейки массива данных *info*. Для пустого стека *S.top=0;*. Все операции над стеком реализуются при помощи операций над массивом *info* данных стека, и при необходимости изменяют индекс *top*.

Динамическая реализация стека. Довольно распространенной является ситуация, когда мы не знаем, сколько элементов может одновременно находиться в стеке, и поэтому не можем использовать массив для реализации стека. В этом случае удобно использовать односвязный линейный список:

```
struct sp
{
   tinfo info;
   sp * next;
};
typedef sp* stek;
stek S;
```

Все операции реализуются при помощи операций над списком. Добавление и удаление элемента в список производятся в голове списка.

2.4 Очередь

Определение 2.3 Очередь – *это структура данных, организованная по принципу "первый пришел – первый ушел" (в англоязычной литературе используется термин FIFO – First In First Out).*

Как правило, структура данных очередь предназначена для временного хранения элементов с целью их последующей обработки. Можно считать, что очередь – это последовательность элементов (возможно пустая), в которой все включения элементов производятся на одном конце последовательности (называемого *концом* очереди), а исключения – на другом ее конце (в *начале* очереди).

В жизни все мы хорошо знакомы с принципом очереди: очереди в магазинах, билетных кассах устроены именно таким образом. Если человек последний встал в очередь, то и уйдет он из нее последним (когда его очередь подойдет); человек, вставший первым в очередь, первым ее покинет. В программировании очереди применяют, например, при буферизированном вводе/выводе, при диспетчеризации задач операционной системой.

Пусть Q – очередь (реализованная некоторым образом), *tinfo* – тип элементов очереди. Для того чтобы работать с очередью, необходимо уметь выполнять следующие основные операции:

1. **void** Create(Queue &Q); – инициализация очереди (создается пустая очередь, в ней пока нет ни одного элемента);
2. **void** Put(Queue &Q, tinfo x); – положить элемент x в очередь;
3. tinfo Get(Queue &Q); извлечь элемент из очереди (элемент удаляется из очереди, значение удаленного элемента – результат данной операции);
4. **bool** Empty(Queue Q); – проверка очереди на пустоту.

В программе очередь можно реализовывать при помощи различных типов данных.

Статическая реализация очереди. Если значение N максимального количества элементов, одновременно находящихся в очереди невелико и заранее известно, то очередь можно реализовать при помощи массива. Преимущество такой реализации – малые затраты по памяти и малое время реализации операций над стеком.

```
const int N=100;
struct Queue
{
  tinfo info[N];
```

```
    int first, last;
};
Queue Q;
```

Индексная переменная *Q.last* указывает на первую свободную после конечного элемента ячейку массива данных *info* очереди (куда будет записываться очередной добавляемый в очередь элемент). Переменная *Q.first* указывает на начало очереди (элемент с данным номер будет первым удален из очереди). Ситуация *Q.first*==*Q.last* означает, что очередь пуста. Все операции над индексами выполняются по $\bmod N$, что позволяет связать элементы массива циклическим образом (за элементом с номером $N-1$ идет элемент с номером 0). Все операции над очередью реализуются при помощи операций над массивом *info* данных очереди, и при необходимости изменяют индексы *Q.first* и *Q.last*.

Динамическая реализация очереди. Когда мы не знаем, сколько элементов может одновременно находиться в очереди, и, следовательно, не можем использовать статический массив для реализации очереди, то удобно использовать для этого односвязный линейный список:

```
struct sp
{
   tinfo info;
   sp * next;
};
struct Queue
{
    sp * first, *last;
}
Queue Q;
```

Все операции реализуются при помощи операций над списком. Добавление элемента производится в конец списка, удаление элемента – из начала списка.

2.5 Деревья

Деревья являются важными и часто используемыми нелинейными структурами данных.

2.5.1 Основные определения

Определение 2.4 Граф – *это пара* (V, E)*, где* V *– непустое множество* вершин *графа,* E *– множество пар* (u, v) *вершин графа, называемых* ребрами графа*. Говорят, что ребро* (u, v) *соединяет вершины* u *и* v*.*

Схематически граф можно изображать в виде точек плоскости (соответствующих вершинам графа), соединенных линиями (соответствующими ребрам графа). Граф называется *ориентированным*, если все ребра граф ориентированы некоторым образом, и *неориентированным*, в противном случае. Для ориентированного графа ребра изображаются в виде стрелок.

Пусть $G = (V, E)$ – некоторый ориентированный граф. Для каждой вершины $v \in V$ *входной степенью* вершины v называется количество входящие в нее ребер. *Выходной степенью* вершины v называется количество выходящих из нее ребер.

Путь, соединяющий вершины u и v – это последовательность вершин $v_0, v_1, \ldots, v_n$ $(n \geq 0)$ такая, что $v_0 = u, v_n = v$ и для любого $i\,(0 \leq i \leq n-1)$ вершины v_i и v_{i+1} соединены ребром.

Граф называется *связным*, если для любой пары вершин графа существует соединяющий их путь.

Дерево – это связный граф без циклов.

Определение 2.5 Двоичное (или бинарное) дерево – *это связный ориентированный граф без циклов такой, что все вершины, кроме одной (которая называется* корнем *дерева) имеют входную степень 1, выходная степень каждой вершины* ≤ 2*. Входная степень корня равна 0. Вершины с выходной степенью 0 называются* листьями *дерева.*

Вершины, не являющиеся корнем, будем называть *внутренними вершинами* дерева.

Утверждение 2.1 *Для каждой внутренней вершины дерева существует ровно один путь, ведущий из корня в эту вершину.*

Длина пути – количество входящих в путь ребер.

Глубина дерева – максимальная длина пути из корня в какой-либо лист.

Схематический двоичное дерево будем изображать таким образом, чтобы корень дерева находился наверху, листья дерева – внизу. Пусть v – некоторая вершина двоичного дерева. Вершины, в которые ведут ребра, исходящие из v будем называть *сыновьями* вершины v. Согласно определению, каждая вершина двоичного дерева имеет не более двух сыновей. Одного из сыновей вершины v будем называть *левым сыном* v (будем обозначать эту вершину $ЛевСын(v)$, на схеме будем изображать его слева от v), другого – *правым сыном* v (будем обозначать его$ПравСын(v)$, на схеме будем изображать его справа от v). Пусть u – сын вершины v. Вершину v будем называть *отцом* вершины u ($Отец(v)$).

Для вершины v все вершины, лежащие на пути из корня в вершину v (не включая ее саму), будем называть *предками* вершины v. Все вершины, лежащие на путях из вершины v в какой-либо лист (не включая саму вершину v) будем называть *потомками* вершины v.

Если D – некоторое дерево, то любая его вершина v определяет некоторое поддерево с корнем в этой вершине v и с внутренними вершинами – потомками вершины v. Пусть v – некоторая вершина двоичного дерева. Поддерево с корнем в $ЛевСын(v)$ будем называть *левым поддеревом вершины* v, поддерево с корнем в $ПравСын(v)$ будем называть *правым поддеревом вершины* v.

Пусть D – некоторое дерево глубины l. Все вершины D можно разбить на *ярусы* $\{0, \ldots, l\}$ следующим образом. К ярусу i ($i = 0, \ldots, l$) отнесем вершины, в которые ведет путь из корня длины i. Согласно утверждению 2.1 каждая вершина относится к одному и только одному ярусу.

Ширина дерева – максимальное количество вершин на ярусе (где максимум берется по всем ярусам дерева).

Дерево будем называть *сбалансированным по высоте*, если для каждой вершины дерева глубины его двух поддеревьев различаются не более чем на единицу.

2.6 Двоичные деревья как структура данных

Деревья являются важными и часто используемыми нелинейными структурами данных. Для простоты мы будем рассматривать только двоичные деревья.

При реализации дерева каждая его вершина *node* содержит информационной

поле, имеющее некоторый тип *tinfo* (будем называть это поле *info*) и два поля, ссылающиеся на левого и правого сына вершины (будем называть эти поля *left* и *right*, соответственно). Тип полей *left* и *right* зависит от реализации типа дерева. Чтобы задать дерево в программе, достаточно задать его корень (будем называть его *root*). Доступ к любой внутренней вершине осуществляется посредством перехода по ребрам дерева. Переменную p будем использовать для указания на текущую вершину (с которой производится работа в данный момент).

Статическая реализация двоичного дерева. Если значение N максимального количества вершин дерева невелико и заранее известно, то дерево можно реализовать при помощи массива. Поля, ссылающиеся на левого и правого сына вершины, в этом случае содержат индексы элементов массива, хранящие левого и правого сыновей. При отсутствии сына в соответствующем поле можно хранить некоторое специальное значение (например, -1). Дерево задается индексом, указывающим на корень дерева. При отсутствии сына

```
struct node
{
  tinfo info;
  int left, right;
};
const  int N=50;
node tree[N];
int root, p;
```

Динамическая реализация двоичного дерева является наиболее часто используемым случаем. В этом случае для каждой вершины дерева поля *left* и *right* содержат указатели на вершины, являющиеся ее левым и правым сыном. При отсутствии сына значение указателя равно NULL. Дерево задается указателем на корень дерева.

```
struct node
{
  tinfo info;
```

```
    node *left, *right;
};
node *root, *p;
```

Всюду ниже в примерах мы будем использовать динамическую реализацию дерева.

2.6.1 Основные операции при работе с деревом

При работе с деревом, как правило, указатель на корень дерева не изменяют. В противном случае мы потеряем доступ к корню, поскольку в нашей реализации переход может быть осуществлен только вниз по дереву (по направлению от корня к листьям). Все манипуляции с деревом осуществляются при помощи текущего указателя p. Некоторые основные операции, осуществляемые при работе с деревом, приведены ниже:

- переход на корень дерева: p=top;
- переход к левому сыну: p=p->left;
- переход к правому сыну: p=p->rigth;
- проверка, является ли текущая вершина листом:
  if ((p->left==NULL)&&(p->right==NULL)) //тогда p– лист;
- проверка, является ли дерево пустым:
  if ((top==NULL)) //тогда дерево пусто.

2.7 Обходы дерева

При работе с деревом, как правило, бывает необходимо тем или иным способом перебирать все его вершины с целью их обработки. Однако данная задача не является столь тривиальной, как для линейных структур данных, таких как линейные списки, поскольку на каждом шаге пути разветвляются. Прохождение дерева в том или ином порядке называется обходом дерева.

Обход дерева – это упорядоченная последовательность вершин дерева, удовлетворяющая следующим требованиям:

- во-первых, в этой последовательности должны присутствовать **все** вершины дерева (т.е. ни одна вершина дерева в процессе обхода не должна быть пропущена);

- во-вторых, каждая вершина дерева присутствует в последовательности ровно один раз.

Только при соблюдении данных требований к обходу дерева мы будем уверены, что ни одна вершина не пропущена и не обработана дважды.

Существуют разные варианты обхода дерева, которые различаются порядком прохода вершин дерева. Среди всех вариантов обхода различают несколько вариантов обхода **в глубину** и варианты обхода **в ширину**.

2.7.1 Обход в глубину

Обход дерева в глубину – это такой обход дерева, при котором вершины перебираются максимально "в глубину" дерева по еще не пройденным путям.

Один из вариантов обхода дерева в глубину – **КЛП**-обход (Корень, Левое, Правое). При **КЛП**-обходе вершины посещаются в следующем порядке:

1. посещается корень дерева;

2. посещается левое поддерево в порядке **КЛП**;

3. посещается правое поддерево в порядке **КЛП**;

Буквы К, Л, П можно переставлять различным образом, получая другие варианты обхода дерева в глубину (например, ЛКП-обход – когда сначала посещаются все вершины левого поддерева в порядке ЛКП, затем посещается корень, затем – все вершины правого поддерева в порядке ЛКП).

Разберем более подробно алгоритм КЛП-обхода дерева. Не рекурсивный алгоритм обхода дерева в глубину использует дополнительную структуру данных – стек для хранения указателей еще не пройденных вершин. Тип информационного поля элемента стека – указатель на вершину дерева:

```
struct node
{
  tinfo info;
```

```
    node *left, *right;
};
//////////////////////////////////////////////////
struct sp
{
    node *info;
    node *next;
};
//////////////////////////////////////////////////
typedef sp* stek;
stek S;

node *root, *p;
```

Алгоритм КЛП-обхода дерева . Выписать значения *info*-поля всех вершин дерева в порядке КЛП.

1. инициализировать стек;
2. если дерево не пусто, положить указатель на корень дерева в стек;
3. пока стек не пуст, выполнять пункты 4–7:
4. достать указатель вершины из стека и сделать эту вершину текущей;
5. выписать значение *info*-поля текущей вершины;
6. если у текущей вершины есть правый сын, положить указатель на правого сына в стек;
7. если у текущей вершины есть левый сын, положить указатель на левого сына в стек;

Использую соответствующие операции для стека (реализованные в виде функций CREATE, PUSH, POP, EMPTY), можно написать функцию, реализующую данный алгоритм. Можно слегка усовершенствовать данный алгоритм, не кладя в стек ту вершину, которая будет на следующем шаге из него извлекаться.

2.7.2 Обход в ширину

Обход дерева в ширину– это такой обход дерева, при котором вершины перебираются в том или ином порядке по ярусам, начиная с нулевого.

Разберем более подробно алгоритм обхода дерева в ширину. Нерекурсивный алгоритм обхода дерева аналогичен приведенному нами выше алгоритму КЛП-обхода дерева с той разницей, что дополнительной структурой данных, используемой для хранения указателей на еще не пройденные вершины, является не стек, а очередь. Аналогично, тип информационного поля элемента очереди – указатель на вершину дерева:

```
struct node
{
  tinfo info;
  node *left, *right;
};
////////////////////////////////////////////////
struct sp
{
  node *info;
  node *next;
};
////////////////////////////////////////////////
struct Queue
{
   sp * first, *last;
}
////////////////////////////////////////////////
Queue Q;

node *root, *p;
```

Алгоритм обхода дерева в ширину. Выписать значения *info*-поля всех вершин дерева в порядке обхода в ширину.

1. инициализировать очередь;
2. если дерево не пусто, положить указатель на корень дерева в очередь;
3. пока очередь не пуста, выполнять пункты 4–7:
4. достать указатель вершины из очереди и сделать эту вершину текущей;
5. выписать значение *info*-поля текущей вершины;
6. если у текущей вершины есть левый сын, положить указатель на левого сына в очередь;
7. если у текущей вершины есть правый сын, положить указатель на правого сына в очередь;

Использую соответствующие операции для очереди (реализованные в виде функций CREATE, PUT, GET, EMPTY), можно написать функцию, реализующую данный алгоритм. Используя счетчики *kol_ cur, kol_ next* для подсчета количества еще не выписанных вершин текущего уровня, и количества вершин следующего уровня, положенных в очередь, можно модифицировать данный алгоритм таким образом, чтобы осуществлялся контроль, когда заканчивается очередной уровень. Таким образом, мы может выписать все вершины дерева в порядке обхода в ширину таким образом, чтобы вершины следующего уровня печатались с новой строки.

2.8 Рекурсивное определение дерева

Можно определить дерево рекурсивным образом. Основываясь на данном рекурсивном определении, можно определять рекурсивные функции для работы с деревом.

Определение 2.6
- *Пустое дерево T – это дерево.*
- *Если дерево T не пусто то существует вершина – корень дерева;*
- *Остальные вершины относятся к непересекающимся множествам (возможно, пустым) T_{left} и T_{right}, которые в свою очередь, являются деревьями.*
- *Из корня дерева T ведет ребро в корень дерева T_{left}, если оно не пусто;*
- *Из корня дерева T ведет ребро в корень дерева T_{right}, если оно не пусто.*

Основываясь на данном рекурсивном определении дерева, динамическую реализацию дерева можно определить рекурсивным образом. Переменная *root* – указатель на дерево определяется следующим образом:

- если дерево пусто, то *root* – это пустой указатель: $root = NULL$;
- если дерево не пусто, то в дереве существует по крайней мере одна вершина – корень дерева, переменная *root* хранит указатель на эту вершину;
- сама вершина-корень дерева представлена в виде структуры из трех полей
  - информационное поле *info*;
  - указатель *left* на левое поддерево корня;
  - указатель *right* на правое подделево корня.

Основываясь на данном рекурсивном определении, можно определять рекурсивные функции для работы с деревом. Как правило, задача для дерева сводится к самой себе, выполняемой на левом и правом поддеревьях и необходимым образом учитывает обработку корневой вершины.

Пример: На рис. 2.3 приведена рекурсивная функция для определения глубины дерева, основанная на следующем рекурсивном алгоритме:

- если дерево пусто, глубина равна -1 (для дерева, состоящего из одной вершины — глубина равна 0);
- если дерево не пусто, необходимо найти глубину левого и правого поддеревьев, взять максимальное из этих значений и прибавить 1 (для ребра, ведущего в поддеревья).

2.9 Дерево двоичного поиска

Будем предполагать, что значения, хранящиеся в вершинах дерева – элементы некоторого линейно упорядоченного множества Z. (Без ограничения общности можно считать, что Z – это множество целых чисел.)

```
typedef int tinfo;
struct node
{
 tinfo info;
 node *left,*right;
};
//////////////////////////////////////////
int Depth(node *root)
{
 if (root==NULL)
    return -1;
 else
     {
      int l=Depth(root->left);
      int r=Depth(root->right);
      if (l>r) return l+1;
      else return r+1;
     }
};
```

Рис. 2.3: Рекурсивная функция нахождения глубины дерева

Определение 2.7 Дерево двоичного (бинарного) поиска – *это двоичное дерево, удовлетворяющее следующему условию. Для любой вершины v значение, находящееся в этой вершине*

- *больше значения любой вершины, находящейся в левом поддереве вершины v и*
- *меньше значения значения любой вершины, находящейся в правом поддереве вершины v.*

Деревья двоичного поиска – эффективный способ организации таблиц информации, позволяющий эффективно выполнять поиск информации, а также операции по внесению изменений в таблицы – удаление, добавление компонент.

Поиск в таких деревьях выполняется значительно эффективнее, чем поиск по списку, поскольку время, затрачиваемое для данной задачи пропорционально глубине дереве. В случае, когда дерево является достаточно сбалансированным, глубина дерева пропорциональна $\log n$, где n – число элементов дерева. Для выполнения поиска в списке из n элементов требуется время, пропорциональное n. (Время здесь измеряется числом пройденных узлов.)

Однако, если дерево имеет не очень “хорошую структуру”, оно может иметь большую глубину, сравнимую с $O(n)$. В частности, дерево может выродиться в линейный список. Этого недостатка лишены “деревья, сбалансированные по высоте”.

Определение 2.8 *Дерево называется* сбалансированным по высоте, *если для каждой его вершины v глубина левого поддерева v и глубина правого поддерева v различаются не более чем на единицу.*

Существуют эффективные алгоритмы поиска, добавления и удаления элементов дерева, которые позволяют сохранять данное его свойство. Мы не будем подробно останавливаться на данном виде деревьев и сосредоточимся просто на деревьях двоичного поиска.

Определим основные операции для работы с деревом двоичного поиска.

Поиск элемента в дереве двоичного поиска. Требуется определить, присутствует ли в дереве двоичного поиска вершина с заданным значением поля *info*. По определение дерева двоичного поиска любому элементу дерева соответствует однозначное его местоположение в дереве согласно его значению. Поэтому, для того, чтобы найти элемент в дереве, достаточно пройти по единственному пути, который определяется значением искомого элемента. На рис.2.4 приведена функция нахождения элемента в дереве двоичного поиска. На рис. 2.5 приведена рекурсивная функция для решения данной задачи.

Добавление элемента в дерево двоичного поиска. Процедура добавления элемента в дерево двоичного поиска должна сохранять данное свойство дерева. Добавляемый в дерево двоичного поиска элемент всегда становится листом. Положение данного листа в дереве определяется значением вставляемого элемента. Стратегия аналогична стратегии поиска: необходимо выполнить поиск. При отсутствии элемента в дереве мы дойдем до некоторой вершины p. Вставляемый элемент становится левым сыном p, если у p не было левого сына и значение вставляемого элемента $<$ значения p, и правым сыном, если у p не было правого сына и значение вставляемого элемента $>$ значения p.

Порождение дерева двоичного поиска. При построении дерева двоичного поиска из заданных элементов достаточно построить дерево, состоящее из одной

```
typedef int tinfo;
struct node
{
 tinfo info;
 node *left,*right;
};
//////////////////////////////////////////

bool Poisk(node *root, tinfo x)
{
 bool b=false;
 node *p=root;
 while ((p!=NULL)&&(b==false))
 {
  if (p->info==x)
     b=true;
  else if (x>p->info)
          p=p->right;
       else p=p->left;
  }
  return b;
}
```

Рис. 2.4: Функция нахождения элемента в дереве двоичного поиска

```
typedef int tinfo;
struct node
{
 tinfo info;
 node *left,*right;
};
//////////////////////////////////////////
bool PoiskRek(node *root, tinfo x)
{
 if (root==NULL)
    return false;
 if (root->info==x)
    return true;
    else if (x>root->info)
         return PoiskRek(root->right, x);
         else
         return PoiskRek(root->left, x);
 }
```

Рис. 2.5: Рекурсивная функция нахождения элемента в дереве двоичного поиска

вершины – корня, поместив в него первый элемент. Для всех оставшихся элементов выполняется задача включения элемента в дерево двоичного поиска.

Удаление элемента из дерева двоичного поиска. Аналогично предыдущей задаче, после удаления элемента из дерева двоичного поиска, данное свойство дерева должно сохраняться. При удалении элемента p из дерева возможны следующие варианты:

- удаляемый элемент p является листом. В этом случае удаление элемента p не представляет сложности. Необходимо иметь указатель на элемент *Отец(p)*. Лист p удаляется, а соответствующая ссылка в элементе *Отец(p)* обнуляется.

- удаляемый элемент p имеет одного сына. В этом случае элемент p удаляется, на его место перемещается его единственный сын.

- удаляемый элемент p имеет двух сыновей. В этом случае можно предложить следующий алгоритм. В левом поддереве p ищется самый правый элемент q. По определению дерева двоичного поиска значение $q >$ значения любого элемента в этом поддереве, но $<$ значения p. Поэтому переписываем значение q вместо значения p и задача сводится к удалению элемента q.

2.10 Деревья арифметических выражений

Двоичные деревья имеют большую сферу приложений. Один из таких примеров – использование деревьев для представления арифметических выражений. Вычисления арифметических выражений, представленных таким образом, а также преобразования над ними, выполняются достаточно эффективно.

Определим класс строк, которые будем называть целочисленными арифметическими выражениями (ЦАВ)следующим образом:

- любой элемент из множества $X = \{0, \ldots, 9, x, y, z\}$ – является ЦАВ. Элементы $\{0, \ldots, 9\}$ будем называть константами, элементы $\{x, y, z\}$ – переменными;

- если α и β – целочисленные арифметические выражения, то строки символов $(\alpha + \beta)$, $(\alpha$ - $\beta)$,$(\alpha$ * $\beta)$,$(\alpha$ / $\beta)$,$(\alpha$ % $\beta)$ являются целочисленными арифметическими выражениями;

- других целочисленнх арифметических выражений нет.

Целочисленные арифметические выражения как строки символов могут быть представлены в памяти машины либо в виде символьных массивов, либо в виде цепочки символов. Однако для выполнения операций над выражениями наиболее удобно представление в виде бинарных деревьев.

Определение 2.9 *Двоичное дерево будем называть* деревом целочисленных арифметических выражений*, если выполняются следующие условия:*

- *любая вершина, не являющаяся листом, имеет ровно двух сыновей;*
- *значение, находящееся в любом листе дерева принадлежит множеству X;*
- *значение, находящееся в любой вершине дерева, не являющейся листом, принадлежит множеству $\{+, -, *, /, \%\}$.*

Представление целочисленного арифметического выражения в виде строки будем называть "выражение-строка". Представление целочисленного арифметического выражения в виде дерева будем называть "выражение-дерево".

Утверждение 2.2 *Для любого арифметического выражения, представленного в виде строки существует однозначное представление в виде дерева и наоборот.*

При работе с деревом арифметических выражений будем выделять следующие операции:

Порождение дерева арифметического выражения. По данному арифметическому выражению, представленному в виде строки (или считываемому из входного потока), построить дерево, представляющее то же самое выражение. Для данной задачи можно предложить алгоритм построения дерева с использованием стека, в котором сохраняем указатели на созданные внутренние вершины (где будет храниться знак операции) до тех пор, пока этот знак операции не будет считан. Знак операции будет считан тогда, когда будет обработан левый аргумент данной операции (который, в свою очередь, тоже может быть арифметическим выражением). Также может быть использован рекурсивный алгоритм для решения данной задачи.

Построение выражения-строки по выражению, представленному в виде дерева. Используем ЛКП-обход дерева, обрабатывая для каждой операции, входящей в выражение сначала левый аргумент, затем – правый, расставляя необходимым образом скобки.

Вычисление арифметического выражения. Вычислить значение выражения, представленного в виде дерева арифметического выражения, поставляя вместо переменных заданные значения. Определим алгоритм вычисления выражения, представленного в виде дерева с уничтожением дерева в процессе вычисления.

1. если дерево-выражение состоит из единственной вершины-корня, то результатом является значение, хранящееся в данной вершине, если это значение – константа, и значение, подставляемое вместо соответствующей переменной, если в корне дерева хранится переменная. Корень дерева уничтожаем.

2. в противном случае используем следующую стратегию. Обозначим через *Знач(v)* значение, хранящееся в вершине v. Будем говорить, что вершина v – это *атом*, если *Знач(v)*$\in \{+, -, *, /, \%\}$, а значения вершин *Знач(ЛевСын(v))* и *Знач(ПравСын(v))* $\in X$. Если вершина v – атом, то

   - заменяем значение вершины v на результат операции *Знач(v)* над аргументами *Знач(ЛевСын(v))* и *Знач(ПравСын(v))*;
   - удаляем вершину *ЛевСын(v)* и обнуляем соответствующую ссылку в вершине v;
   - удаляем вершину *ПравСын(v)* и обнуляем соответствующую ссылку в вершине v;

   Совершаем ЛПК-обход дерева-выражения, выполняя соответствующие преобразования над вершинами, являющимися атомами до тех пор, пока дерево не будет содержать лишь одну вершину-корень. В этом случае поступаем согласно пункту 1. Обход дерева может быть реализован с использованием стека или рекурсивно.

Преобразования арифметического выражения. Преобразования над арифметическим выражением, представленным в виде дерева, выполняются при помощи соответствующих операций над деревом.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Павловская Т.А. Программирование на языке высокого уровня. – СПб.:Питер, 2007. - 461с.

2. Мартынов Н.Н. Программирование для Windows на C/C++. Том 1. - М.:ООО “Бином-Пресс”, 2004г. – 528 с.

3. Карпов Б., Баранова Т. C++. Специальный справочник (2-е издание). - СПб.:Питер,2005. - 381с.

4. Касьянов В.Н., Сабельфельд В.К., Сборник заданий по практикуму на ЭВМ. Учебное пособие для вузов. – М.:Наука, 1986. - 272с.

5. А.Ахо, Дж.Хопкрофт, Дж.Ульман. Построение и анализ вычислительных алгоритмов. – М:Изд-во МИР. - 1979г.