

КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Конспекты лекций по курсу

Информатика

для студентов ИВМиИТ 1 семестр 1 курс

составитель Гайнутдинова А.Ф.

Казанский федеральный университет

2016

Оглавление

| | | |
|----------|---|----------|
| 1 | Основные конструкции языка C++ | 5 |
| 1.1 | Программирование | 5 |
| 1.2 | Разработка алгоритма | 6 |
| 1.2.1 | Основные этапы разработки алгоритма | 7 |
| 1.2.2 | Запись алгоритма на языке программирования | 10 |
| 1.2.3 | Выбор представления данных | 11 |
| 1.2.4 | Сложность алгоритма | 12 |
| 1.2.5 | Отладка | 13 |
| 1.2.6 | Эксплуатация программы | 15 |
| 1.3 | Блок-схемы | 15 |
| 1.4 | Структурное программирование | 17 |
| 1.4.1 | Методы структурирования блок схем | 17 |
| 1.5 | Типы данных | 19 |
| 1.6 | Переменные | 22 |
| 1.7 | Операторы ветвления | 23 |
| 1.7.1 | Условные операторы | 23 |
| 1.7.2 | Оператор выбора | 25 |
| 1.8 | Операторы передачи управления | 27 |
| 1.8.1 | Оператор безусловного перехода goto | 27 |
| 1.8.2 | Оператор break | 27 |
| 1.8.3 | Оператор continue | 27 |
| 1.9 | Операторы цикла | 28 |
| 1.9.1 | Оператор цикла while | 28 |
| 1.9.2 | Оператор цикла do while | 29 |
| 1.9.3 | Оператор цикла for | 29 |
| 1.10 | Понятие потока. Стандартные входной и выходной потоки | 30 |

| | | |
|--------|--|----|
| 1.10.1 | Буферизация ввода/вывода | 31 |
| 1.10.2 | Стандартный входной поток. | 32 |
| 1.10.3 | Стандартный выходной поток. | 34 |
| 1.11 | Массивы | 35 |
| 1.11.1 | Многомерные массивы | 39 |
| 1.12 | Некоторые аспекты математической логики в программировании . . | 39 |
| 1.12.1 | Стратегии вычисления кванторов | 40 |
| 1.13 | Реализация операций над множествами с использованием массивов . | 42 |
| 1.13.1 | Операции над множествами | 43 |
| 1.13.2 | Представление множества при помощи массива | 44 |
| 1.13.3 | Пример: Алгоритм “Решето Эратосфена” | 44 |
| 1.14 | Представление полиномов при помощи массивов. Схема Горнера. . . . | 45 |
| 1.14.1 | Вычисление полинома в точке. Схема Горнера | 46 |
| 1.15 | Символьный тип данных | 47 |
| 1.15.1 | Преобразования типов из int в char и из char в int | 47 |
| 1.16 | Строки | 49 |
| 1.17 | Файлы | 53 |
| 1.18 | Файловые потоки | 54 |
| 1.19 | Текстовые файлы | 55 |
| 1.20 | Линейные упорядоченные последовательности | 59 |
| 1.20.1 | Поиск элемента | 60 |
| 1.20.2 | Включение одной последовательности в другую | 61 |
| 1.20.3 | Объединение двух упорядоченных последовательностей | 62 |
| 1.20.4 | Пересечение двух упорядоченных последовательностей | 63 |
| 1.20.5 | Разность двух упорядоченных последовательностей | 65 |
| 1.21 | Простейшие алгоритмы сортировки последовательности | 65 |
| 1.21.1 | Алгоритм сортировки методом обмена пар | 67 |
| 1.21.2 | Алгоритм сортировки методом нахождения локальных экстре- мумов | 68 |
| 1.21.3 | Алгоритм сортировки вставкой | 69 |
| 1.21.4 | Алгоритм сортировки фон-Неймана (сортировка слиянием) . . | 70 |
| 1.21.5 | Сравнение алгоритмов сортировки | 70 |
| 1.22 | Функции | 71 |
| 1.22.1 | Описание функции | 72 |

| | | |
|---------|---|----|
| 1.22.2 | Вызов функции | 73 |
| 1.22.3 | Формальные и фактические параметры | 74 |
| 1.22.4 | Локальные и глобальные переменные | 74 |
| 1.22.5 | Передача параметров по значению и по ссылке | 75 |
| 1.22.6 | Входные и выходные параметры | 76 |
| 1.22.7 | Параметры по умолчанию | 77 |
| 1.22.8 | Модифицированное тело функции | 77 |
| 1.22.9 | Перегрузка функций | 77 |
| 1.22.10 | inline -функции | 78 |

| | |
|-------------------|-----------|
| Литература | 79 |
|-------------------|-----------|

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 деления 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 проектирование. 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 обязательно 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, разбиение 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 mnemonicными.

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 значений 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). Диапазон 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 областью 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 **switch** для обеспечения выхода в точку программы, 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_2* 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 программирования 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 приемнику.

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 $\langle a_1, a_2, \dots, a_m \rangle$ – содержимое входного потока, где $\langle a_1, a_2, \dots, a_m \rangle$ – 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 $\langle a_2, \dots, a_m \rangle$.

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:

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

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 >> name1 >> name2 >> ... >> namek;`

аналогична последовательности команд

`cin >> name1;`

`cin >> name2;`

...

`cin >> namek;`

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 $\langle a_1, a_2, \dots, a_m \rangle$ – содержимое выходного потока. Тогда после выполнения команды вывода

- содержимое выходного потока становится $\langle a_1, a_2, \dots, a_m, x \rangle$,

- 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 >> выражение1 >> выражение2 >> ... >> выражениеk;`

аналогична последовательности команд

`cout >> выражение1;`

`cout >> выражение2;`

...

`cout >> выражениеk;`

Некоторые форматирующие символы:

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

Пример: Ввести два целых числа n и m ($n < m$) и вывести на экран все числа $> n$ и $< m$. 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:

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

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 “**А**”- формула и “**Е**”- формула и приведем стратегии вычисления данных формул.

“А”- формула: $\forall x B(x)$ – для всех x выполняется свойство $B(x)$

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

$$\forall x B(x) \Leftrightarrow B(x_1) \&\& B(x_2) \&\& \dots \&\& B(x_n)$$

Можно задать рекуррентное определение квантора:

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

Это определение дает следующую стратегию вычисления “**А**”- 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 $b = \text{FALSE}$; | 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. |

“Е”- формула: $\exists x B(x)$ – существует x , для которого выполняется свойство $B(x)$

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

$$\exists x B(x) \Leftrightarrow B(x_1) \parallel B(x_2) \parallel \dots \parallel B(x_n)$$

Можно задать рекуррентное определение квантора:

$$b_0 = false;$$

$$b_{i+1} = b_i \parallel B(x_{i+1})$$

Это определение дает следующую стратегию вычисления **“Е”- 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 $x = (x_0, \dots, x_{N-1})$ в порядке неубывания. Задача может быть сформулирована как проверка \forall -свойства следующим образом:
 $(\forall i \in \{0, \dots, N-2\} : x[i] \leq x[i+1]);$
- Поиск заданного элемента y в массиве $x = (x_0, \dots, x_{N-1})$. Задача может быть сформулирована как проверка \exists -свойства следующим образом:
 $(\exists i \in \{0, \dots, N-1\} : x[i] == y);$
- Проверить равенство двух массивов $x = (x_0, \dots, x_{N-1})$ и $y = (y_0, \dots, y_{N-1})$. Задача может быть сформулирована как проверка \forall -свойства следующим образом:
 $(\forall i \in \{0, \dots, N-1\} : x[i] == y[i]);$

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 $y = (y_0, \dots, y_{M-1})$ в массив $x = (x_0, \dots, x_{N-1})$ как сплошной фрагмент ($M < N$). Задача может быть сформулирована следующим образом:
 $(\exists i \in \{0, N-M\} : (\forall j \in \{0, \dots, M-1\} : y[j] == x[i+j]));$
- Проверка периодичности числового массива $x = (x_0, \dots, x_{N-1})$. Задача может быть сформулирована следующим образом:
 $(\exists r \in \{1, \dots, n/2\} : (\forall i \in \{0, \dots, N-r-1\} : x[i] == x[i+r]));$

1.13 Реализация операций над множествами с использованием массивов

Пусть $S = \{s_0, s_1, \dots, s_{n-1}\}$ – 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 $S = \{0, 1, \dots, 9\}$. Примеры множеств над S :

$A = \emptyset$ – пустое множество,

$A = \{5, 3, 9\} = \{3, 5, 9\} = \{9, 3, 5\}$,

$A = \{1, 3, 5, 7, 9\}$,

$A = \{0, 1, \dots, 9\}$.

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 $A_1 \cup A_2$;
- Пересечение множеств $A_1 \cap A_2$;
- 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 ($a = (a_0, a_1, \dots, a_{n-1})$) 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, состоящий только из единиц, задает множество, совпадающее с базовым множеством S . Тогда можно сопоставить теоретико-множественные операции и операции над характеристическими векторами.

$$x \in A \quad \Leftrightarrow a[x] == 1;$$

$$A \subseteq B \quad \Leftrightarrow \forall x a[x] == 1 \Rightarrow b[x] == 1;$$

$$C = A \cup B \quad \Leftrightarrow \forall x c[x] = a[x] || b[x];$$

$$C = A \cap B \quad \Leftrightarrow \forall x c[x] = a[x] \&\& b[x];$$

$$C = A \setminus B \quad \Leftrightarrow \forall x c[x] = a[x] \&\& !b[x];$$

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 $N - 1$

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

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

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

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 $P = (p_0, \dots, p_{N-1})$. 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 вычисления очередной степени 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

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

Используя такое представление полинома можно предложить другой алгоритм *Alg2* 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 данный 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 некой 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 $x = 7$;

Обратное преобразование из **int** в **char** выполняется с использованием тех же соображений. Например, код символа '5' ровно на 5 больше кода символа '0', поэтому для получения 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,
                       //переведа 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 функций 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 присутствию 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 доступом 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,

потокковая 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 потокковая 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.flush*();

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

6. Заккрытие 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, \dots, a_{n-1} называется упорядоченной в порядке неубывания, если для любых $i = 0, \dots, n-2$ 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 $a_0, a_1 \dots, a_{n-1}$ ($a_0 \leq a_1 \leq \dots \leq a_{n-1}$) и пусть дан элемент y . Определить имеется ли в последовательности 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 $a_0, a_1 \dots, a_{n-1}$ 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 $a = (a_0, a_1 \dots, a_{n-1})$. 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 $a : a_0, a_1, \dots, a_{n-1}$ и $b : b_0, b_1, \dots, b_{m-1}$. Верно ли, что $a \subseteq b$ (a включается в b):

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

В случае неупорядоченных последовательностей для каждого элемента $a_i, i = 0, \dots, n - 1$ 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 имеем дело со структурой данных с последовательным доступом, то для каждого $a_i (i = 0, \dots, n - 1)$ 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 $S(Alg5) = n + m$. 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 $a : a_0, a_1 \dots, a_{n-1}$ и $b : b_0, b_1 \dots, b_{m-1}$. Построить упорядоченную последовательность $c : c_0, c_1 \dots, c_{k-1}$, 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 : $c = a \cup b$.

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 $a : a_0, a_1 \dots, a_{n-1}$ и $b : b_0, b_1 \dots, b_{m-1}$. Построить упорядоченную последовательность $c : c_0, c_1 \dots, c_{k-1}$, 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 : $c = a \cap b$.

В случае неупорядоченных последовательностей алгоритм имеет сложность $n * m$, поскольку для каждого элемента a_i , ($i = 0, \dots, n - 1$) 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 $a : a_0, a_1 \dots, a_{n-1}$ и $b : b_0, b_1 \dots, b_{m-1}$. Построить упорядоченную последовательность $c : c_0, c_1 \dots, c_{k-1}$, 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 $R = R_1, R_2, \dots, R_n$ снабженных, соответственно, ключами K_1, K_2, \dots, K_n . Можно считать, что каждый элемент последовательности $R_i = \langle K_i, I_i \rangle$, 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, \dots, R_n таким образом, чтобы в результирующей последовательности элементы располагались в порядке неубывания значений их ключей, т.е. $K_1 \leq k_2 \leq \dots \leq K_n$ (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 $1, 2, \dots, n$, что $K_{\pi(1)} \leq K_{\pi(2)} \leq \dots \leq K_{\pi(n)}$. При этом сами элементы R_1, R_2, \dots, 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 $\langle R, A \rangle$, 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 убывания, т.е. чтобы для любых $i = 1, \dots, N - 1$ 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 $a[i]$ ($i = 1, \dots, N - 1$). 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 позицию). Его исключаем из рассмотрения и снова запускаем процесс на с подмассиве $a[0] \dots a[N - 2]$. 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 ($j = 1, \dots, N - 1$) элемент $a[j]$ вставляется в уже упорядоченную последовательность $a[0], \dots, a[j - 1]$ следующим образом. Элемент $a[j]$ копируется во вспомогательную переменную s , тем самым освобождая j -ую ячейку массива. Упорядоченный подмассив $a[0], \dots, a[j - 1]$ просматривается с конца. Все его элементы, большие элемента $s = a[j]$ (и которые, следовательно, должны стоять правее его), сдвигаются на разряд вправо. На освободившееся место записывается элемент s .

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 сливанием)

Данный алгоритм основан на идее слияния двух упорядоченных последовательностей.

Алгоритм: Пусть, для простоты изложения длина сортируемой последовательности a равна $2 \cdot k$ 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 $2 \cdot k$. 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, оставшая 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* ($i = 1, \dots, k$) 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 функцию *по ссылке* оформляется в списке формальных параметров следующим образом:

type_ ¶m_

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 по 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 отсутствия 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.

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.