МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ УКРАИНЫ
ОДЕССКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ
имени И.И. МЕЧНИКОВА

Институт математики, экономики и механики
Кафедра математического обеспечения компьютерных систем

Т.И. Петрушина, И.Н. Лисицына

Программирование

***Пособие по учебной практике
для студентов 1-го курса
специальности «Прикладная математика»***

Одесский национальный
университет
имени И.И. Мечникова
2010

Программирование. Пособие по учебной практике для студентов 1-го курса специальности «Прикладная математика».

Авторы-составители:

Т. И. Петрушина, канд. физ.-мат. наук, доцент,
зав. кафедрой математического обеспечения компьютерных систем

И. Н. Лисицына, старший преподаватель
кафедры математического обеспечения компьютерных систем

Рецензенты:

В. С. Макогон, кандидат физ.-мат. наук, доцент
кафедры математического обеспечения компьютерных систем

В. Г. Пенко, кандидат технических наук, доцент
кафедры математического обеспечения компьютерных систем

Рекомендовано к изданию Учёным советом
Института математики, экономики и механики ОНУ.
Протокол № 1 от 9 октября 2009 года.

© Одесский национальный университет имени И. И. Мечникова

Оглавление

Введение..4
Выполнение студентами всех заданий, предложенных в методическом пособии, с учетом предъявленных требований, безусловно, свидетельствует об отличном усвоении материала курса.Задание по учебной практике №1. Работа с массивами 4
Задание по учебной практике №1. Работа с массивами 5
Чистка циклов 5
Постановка задачи... 7
Содержание отчета... 7
Варианты задания... 7
Задание по учебной практике №2. Численные методы решения уравнений. 10
Описание численных методов решения уравнений....................................... 10
Постановка задачи... 16
Содержание отчета... 16
Варианты задания... 17
Задание по учебной практике №3. Работа со строками. Связное хранение строк. 18
Цепочки 18
Постановка задачи..21
Содержание отчета... 22
Варианты задания... 22
Задание по учебной практике № 4. Реализация методов сортировок...................................24
Алгоритмы сортировки...24
Постановка задачи..31
Содержание отчета..32
Варианты задания..32
Методические указания для выполнения задания №4 33
Задание по учебной практике № 5. Программирование лексического анализатора...........34
Лексический анализ 34
Постановка задачи...36
Содержание отчета..38
Варианты задания..38
Методические указания для выполнения задания №5. 39
Приложение. Некоторые средства языка С++, необходимые для выполнения заданий учебной практики. 40

Введение

Цель учебной практики состоит в применении знаний и навыков, полученных студентами в ходе изучения курса «Программирование» для выполнения практических заданий.

Набор заданий по учебной практике охватывает основные разделы курса и требует от студентов умения самостоятельной работы с математической литературой.

Каждое задание включает в себя сжато и ясно изложенный необходимый теоретический материал, дополнительные указания детализируют поставленную перед студентами задачу.

Поскольку задачей курса является, в первую очередь, знакомство с основными методами и приемами программирования, а не просто изучение того или иного языка программирования, язык для выполнения заданий не оговаривается. Выбор языка диктуется требованиями, предъявляемыми к заданиям.

Выполнение студентами всех заданий, предложенных в методическом пособии, с учетом предъявленных требований, безусловно, свидетельствует об отличном усвоении материала курса.

Задание по учебной практике №1.

Работа с массивами

Чистка циклов

Довольно часто в практике программирования возникают задачи вычисления значений различных конечных сумм: полиномов, отрезков степенных рядов, коэффициенты которых заданы явными выражениями в зависимости от номера члена, и т. п. С основными приемами, используемыми при программировании таких задач, познакомимся на следующем примере.

Вычислить сумму отрезка степенного ряда для sin x:

$$\sum_{i=0}^{n}(-1)^{i}\frac{x^{2i+1}}{(2i+1)!}$$

Будем считать, что номер n последнего члена суммы не задан явно, а определяется требованием, чтобы последний член u_n не превосходил по абсолютной величине заданное число ε. Здесь

$$u_i=(-1)^i\frac{x^{2i+1}}{(2i+1)!}$$

Полезно рассмотреть отношение

$$\frac{u_i}{u_{i-1}}=\frac{(-1)^i x^{2i+1}(2i-1)!}{(2i+1)!(-1)^{i-1}x^{2i-1}}=-\frac{x^2}{2i(2i+1)},$$

откуда

$$u_i=-u_{i-1}\frac{x^2}{2i(2i+1)}$$

Здесь множитель $-x^2$ (обозначим его через v) не зависит от i, а знаменатель $2i(2i+1)$ вычисляется в зависимости от i сравнительно просто - без циклов. Алгоритм вычисления суммы, в котором цикл вычисления самой суммы совмещен с циклами, необходимыми для вычисления x^{2i+1}, $(2i+1)!$ и даже $(-1)^i$, оказывается таким:

1. $i := 0$, $u := x$, $s := x$, $v := —x^2$.

2. $i: = i + 1$, $u = \dfrac{uv}{2i(2i+1)}$, $s: = s + u$.

3. *Если* $|u| > \varepsilon$, то *перейти на* 2.

Следует обратить внимание на то, что вычисление $v := - x^2$ вынесено в подготовительную часть цикла (шаг 1), чтобы не повторять его каждый раз заново. Перейти от алгоритма к машинной программе теперь уже не трудно.

Заметим, что из шести команд, затрачиваемых на вычисление u, четыре команды уходят на вычисление знаменателя выражения для u (в это число мы включили и команду вычисления очередного значения i, поскольку нигде, кроме этого знаменателя, значение переменной i не используется). Можно упростить вычисление знаменателя, по существу, тем же самым приемом. Обозначим этот знаменатель через r_i,:

$$r_i=2i(2i+1)=4i^2+2i$$

и постараемся выразить значение r_i,- через r_{i-1}. Отношение r_i к r_{i-1} оказывается сложным, поэтому попробуем рассмотреть разность

$$d_i = r_i - r_{i-1} = 4i^2+2i-4(i - 1)^2 - 2(i - 1)=8i-2,$$

откуда

$$r_i=r_{i-1}+d_i,$$

а di выражается через di-1 уже совсем просто:

$$d_i=d_{i-1}+8.$$

Таким образом, очередное значение r_i вычисляется через предыдущее с помощью всего двух операций (одна из них вычисляет d_i через d_{i-1}), а надобность в переменной i отпала. Чтобы начать вычисления, нужно знать начальные значения переменных r и d (при i=0). Они равны, соответственно 0 и -2. Итак, еще один вариант алгоритма может быть записан в следующем виде.

1' $d := -2$, $r : = 0$, $u := x$, $s := x$, $v :=-x^2$.

2' $d :=d + 8$, $r := r + d$, $u := uv/r$, $s := s + u$.

3' *Если* $|u| > \varepsilon$, то *перейти на* 2'.

Здесь очень важна правильная последовательность присваиваний на шаге 2'.

Второй вариант программы короче первого всего на одну команду. Ради этого не стоило бы затевать переделку. Однако повторяющаяся часть программы (блоки 2' и 3') стала короче на две команды, так что на выполнение всей программы затрачивается *7n*

+ *6* операций (*n* — число повторений цикла) вместо *9n + 5* операций, идущих на выполнение первого варианта программы. Уже при $n = 3$ экономия времени выполнения программы составляет около 15%, а такая экономия оправдывает затраченные усилия. Важным аргументом является и то, что вещественная арифметика – приближенная, а уменьшение количества операций уменьшает и ошибку округления.

Сокращение повторяющейся части цикла за счет устранения из нее одинаковых вычислений называется *чисткой* цикла. Чистка циклов — это один из важнейших способов ускорения работы программы.

Постановка задачи

1. Получить элементы квадратной матрицы $A = \left\{ a_{ij} \right\}_{i,j=1}^{n}$, где $a_{ij} = f(i, j, \varphi, \mu)$, а $f(i, j, \varphi, \mu)$ – заданная функция, зависящая от параметров φ и μ.
2. Из матрицы *A* по заданному правилу получить компоненты вектора $D = \left\{ d_i \right\}_{i=1}^{n}$.
3. Вычислить значения величины *B* по элементам вектора *D*.

Для вычисления каждого из объектов *A*, *B* и *D* должна быть написана соответствующая подпрограмма. При необходимости для решения задачи возможно использование вспомогательных функций. Значения *n*, φ и μ выбираются самостоятельно и задаются в программе в виде констант.

В функции для вычисления элементов матрицы должна быть реализована **чистка циклов**.

Содержание отчета

1. Постановка задачи (конкретный вариант).
2. Выкладки, необходимые для чистки цикла.
3. Спецификации подпрограмм, необходимых для решения задачи.
4. Результаты решения.
5. Текст программы на электронном носителе.

Варианты задания

Для определения варианта задания следует по номеру *k* (порядковый номер студента в списке), выбрать номер правила для вычисления элементов матрицы равным остатку от деления *k* на 6, номер правила для вычисления элементов вектора D равным

остатку от деления k на 10 и номер правила для вычисления числа B равным остатку от деления k на 8.

Правила для получения матрицы A:

0. $2^{i-j}19\cos(2i-j+\varphi)\mu e^{-\varphi^2(i-j)^2}$
1. $\dfrac{2-(j-3)^2}{i!}(j-5.7)\cos(\mu\varphi^i)$
2. $2\left(\dfrac{j}{2}-3.1\right)(-2)^i(i-3.9)tg(\mu^i-\varphi)$
3. $(|j-3|-1.3)(6.5-j)2^{j-1}(2|3.3-i|-1.5)\sin(\mu+i\varphi)$
4. $(|3.7-j|-2)(i-4.3)2^{-i}ctg(\varphi-\mu^{-1})$
5. $\dfrac{2^i}{(i-2.5)^{i^2}}\left(2-(j-5)^2\right)\sin\left(\dfrac{\varphi}{\mu^{2i}}\right)$

Правила получения вектора D по матрице A:

0. d_i - скалярное произведение i-ой строки матрицы A на столбец, содержащий первый по порядку наибольший элемент данной строки.
1. В качестве вектора D принять главную диагональ матрицы A , преобразованной следующим образом: в начале каждой строки должны быть собраны ее неотрицательные элементы, а в конце отрицательные элементы (с сохранением порядка следования тех и других элементов).
2. В матрице A найти первую по порядку строку с наибольшей суммой ее элементов и в качестве компонентов вектора D принять эту строку.
3. Компоненты вектора D – средние арифметические значения элементов строк матрицы A.
4. В качестве d_i принять скалярное произведение строки с наибольшим количеством неотрицательных элементов на i-ый столбец.
5. В качестве d_i принять скалярное произведение i-ой строки на столбец с минимальным количеством положительных элементов.
6. В качестве d_i принять $\left|\max\limits_{0\le j<n}\{a_{ij}\}\right|-\left|\min\limits_{0\le j<n}\{a_{ij}\}\right|, \quad i=0,1,\ldots,n-1$.

7. В матрице A найти первую по порядку строку с максимальной суммой модулей ее элементов. Вектор D получить из найденной строки циклическим сдвигом ее элементов на две позиции влево.
8. В качестве d_i принять число отрицательных элементов в i-ой строке.
9. В качестве d_i принять произведение квадратов тех элементов i-ой строки, модули которых принадлежат отрезку [1, 1.5].

Правила вычисления величины B по вектору D:

0. $\sum_{i=0}^{n-1} d_i d_{n-i-1}$
1. $\prod_{i=0}^{n-1} \left(d_i + d_{n-i-1}\right)$
2. $\max_{0 \le i < n-1} \left\{ |d_{i+1}| - |d_i| \right\}$
3. $\left(d_0 + d_1 + d_2\right) d_1 + \left(d_1 + d_2 + d_3\right) d_2 + \cdots + \left(d_{n-3} + d_{n-2} + d_{n-1}\right) d_{n-2}$
4. $\dfrac{1 + \sum_{d_i = 1} d_i}{2 + \sum_{d_i > 1} d_i}$
5. $\max_{0 <= i < n} \left\{ |d_i| \right\}$
6. $\sqrt{d_0 d_1 d_2 \cdots d_{n-1}}$
7. $\prod_{i=0}^{n-2} \left(\dfrac{1}{|d_i| + 1} + d_{i+1} \right)$

Литература

1. Лавров С. С. Введение в программирование. -Издательство: М.: Наука, 1977. -368с.

Задание по учебной практике №2.
Численные методы решения уравнений.

Описание численных методов решения уравнений

Общая постановка задачи. Найти действительные корни уравнения $f(x)=0$, где $f(x)$ - алгебраическая или трансцендентная функция.

Точные методы решения уравнений подходят только к узкому классу уравнений (квадратные, биквадратные, некоторые тригонометрические, показательные, логарифмические).

В общем случае решение данного уравнения находится приближённо в следующей последовательности:

1) отделение (локализация) корня;
2) приближённое вычисление корня до заданной точности.

Отделение корня. Отделение действительного корня уравнения $f(x)=0$ - это нахождение отрезка $[a;b]$, в котором лежит только один корень данного уравнения. Такой отрезок называется отрезком изоляции (локализации) корня.

Наиболее удобным и наглядным является графический метод отделения корней:

1) строится график функции $y=f(x)$, и определяются абсциссы точек пересечения этого графика с осью Ox, которые и являются корнями уравнения $f(x)=0$;

2) если $f(x)$- сложная функция, то её надо представить в виде $f(x)=\varphi_1(x)-\varphi_2(x)$ так, чтобы легко строились графики функций $y=\varphi_1(x)$ и $y=\varphi_2(x)$. Так как $f(x)=0$, то $\varphi_1(x)=\varphi_2(x)$. Тогда абсциссы точек пересечения этих графиков и будут корнями уравнения $f(x)=0$.

Уточнение корня.

Если искомый корень уравнения $f(x)=0$ отделён, т.е. определён отрезок $[a;b]$, на котором существует только один действительный корень уравнения, то далее необходимо найти приближённое значение корня с заданной точностью.

Такая задача называется задачей уточнения корня.

Уточнение корня можно производить различными методами:

1) методом половинного деления (бисекции);
2) методом простых итераций;
3) методом хорд (секущих);
4) методом касательных (Ньютона);
5) комбинированным методом.

Метод половинного деления (бисекции).

Отрезок изоляции корня можно уменьшить путём деления его пополам.

Такой метод можно применять, если функция $f(x)$ непрерывна на отрезке $[a;b]$ и на его концах принимает значения разных знаков, т.е. выполняется условие

$$f(a)\cdot f(b)<0. \qquad (1)$$

Разделим отрезок $[a;b]$ пополам точкой $c_1=\frac{a+b}{2}$, которая будет приближённым значением корня $\bar{x}$.

Для уменьшения погрешности приближения корня уточняют отрезок изоляции корня. В этом случае продолжают делить отрезки, содержащие корень, пополам.

Из отрезков $[a;c_1]$ и $[c_1;b]$ выбирают тот, для которого выполняется неравенство (1).

В нашем случае это отрезок $[c_1;b]$, где $f(c_1)\cdot f(b)<0$.

Далее повторяем операцию деления отрезка пополам, т.е. находим $c_2=\frac{c_1+b}{2}$ и так далее до тех пор, пока не будет достигнута заданная точность ε. Т.е. до тех пор, пока не перестанут изменяться сохраняемые в ответе десятичные знаки или до выполнения неравенства $|c_{i+1}-c_i|<2\varepsilon$.

Достоинство метода: простота (достаточно выполнения неравенства (1)).

Недостаток метода: медленная сходимость результата к заданной точности.

Метод хорд (секущих).

Этот метод применяется при решении уравнений вида $f(x)=0$, если корень уравнения отделён, т.е. $x \in [a;b]$ и выполняются условия:

1) $f(a)\cdot f(b)<0$ (функция $f(x)$ принимает значения разных знаков на концах отрезка $[a;b]$);

2) производная $f'(x)$ сохраняет знак на отрезке $[a;b]$ (функция $f(x)$ либо возрастает, либо убывает на отрезке $[a;b]$).

Первое приближение корня находится по формуле: $x_1 = a - \dfrac{(b-a)\cdot f(a)}{f(b)-f(a)}$.

Для следующего приближения из отрезков $[a;x_1]$ и $[x_1;b]$ выбирается тот, на концах которого функция $f(x)$ имеет значения разных знаков.

Тогда второе приближение вычисляется по формуле:

$x_1 = a - \dfrac{(x_1-a)\cdot f(a)}{f(x_1)-f(a)}$, если $\bar{x} \in [a;x_1]$ или $x_2 = x_1 - \dfrac{(b-x_1)\cdot f(x_1)}{f(b)-f(x_1)}$, если $\bar{x} \in [x_1;b]$.

Вычисления продолжаются до тех пор, пока не перестанут изменяться те десятичные знаки, которые нужно оставить в ответе.

Метод касательных (Ньютона).

Этот метод применяется, если уравнение $f(x)=0$ имеет корень $\bar{x} \in [a;b]$, и выполняются условия:

1) $f(a)\cdot f(b)<0$ (функция принимает значения разных знаков на концах отрезка $[a;b]$);

2) производные $f'(x)$ и $f''(x)$ сохраняют знак на отрезке $[a;b]$ (т.е. функция $f(x)$ либо возрастает, либо убывает на отрезке $[a;b]$, сохраняя при этом направление выпуклости).

На отрезке $[a;b]$ выбирается такое число x_0, при котором $f(x_0)$ имеет тот же знак, что и $f''(x_0)$, т. е. выполняется условие $f(x_0)\cdot f''(x_0)>0$. Таким образом, выбирается точка с абсциссой x_0, в которой касательная к кривой $y=f(x)$ на отрез-

ке $[a;b]$ пересекает ось Ox. За точку x_0 сначала удобно выбирать один из концов отрезка.

Первое приближение корня определяется по формуле: $x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$.

Второе приближение корня определяется по формуле: $x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$.

Вычисления ведутся до совпадения десятичных знаков, которые необходимы в ответе, или при заданной точности ε - до выполнения неравенства $|x_n - x_{n-1}| < \varepsilon$.

Достоинства метода: простота, быстрота сходимости.

Недостатки метода: вычисление производной и трудность выбора начального положения.

Комбинированный метод хорд и касательных.

Если выполняются условия:

1) $f(a) \cdot f(b) < 0$,

2) $f'(x)$ и $f''(x)$ сохраняют знак на отрезке $[a;b]$,

то приближения корня $\bar{x} \in [a;b]$ уравнения $f(x) = 0$ по методу хорд и по методу касательных подходят к значению этого корня с противоположных сторон. Поэтому для быстроты нахождения корня удобно применять оба метода одновременно. Т.к. один метод даёт значение корня с недостатком, а другой – с избытком, то достаточно легко получить заданную степень точности корня.

Схема решения уравнения методом хорд и касательных

1) Вычислить значения функции $f(a)$ и $f(b)$.

2) Проверить выполнение условия $f(a) \cdot f(b) < 0$. Если условие не выполняется, то неправильно выбран отрезок $[a;b]$.

3) Найти производные $f'(x)$ и $f''(x)$.

4) Проверить постоянство знака производных на отрезке $[a;b]$. Если нет постоянства знака, то неверно выбран отрезок $[a;b]$.

Для метода касательных выбирается за x_0 тот из концов отрезка $[a;b]$, в котором выполняется условие $f(x_0) \cdot f''(x_0) > 0$, т.е. $f(x_0)$ и $f''(x_0)$ одного знака.

Приближения корней находятся:

а) по методу касательных: $x_{11} = x_0 - \dfrac{f(x_0)}{f'(x_0)}$,

б) по методу хорд: $x_{12} = a - \dfrac{(b-a)f(a)}{f(b)-f(a)}$.

Вычисляется первое приближение корня: $\xi_1 = \dfrac{x_{11}+x_{12}}{2}$.

Проверяется выполнение условия: $|\xi_1 - x_{11}| < \varepsilon$, где ε - заданная точность.

Если условие не выполняется, то нужно продолжить применение метода по схеме 1-8.

В этом случае отрезок изоляции корня сужается и имеет вид $[x_{11}; x_{12}]$. Приближённые значения корня находятся по формулам:

$$x_{21} = x_{11} - \frac{f(x_{11})}{f'(x_{11})} \text{ и } x_{22} = x_{11} - \frac{(x_{12}-x_{11})f(x_{11})}{f(x_{12})-f(x_{11})}.$$

Вычисления продолжаются до тех пор, пока не будет найдено такое значение ξ, при котором x_{n1} и x_{n2} совпадут с точностью ε.

Метод простых итераций. Предположим, что уравнение $f(x)=0$ при помощи некоторых тождественных преобразований приведено к виду $x=\varphi(x)$.

Заметим, что такое преобразование можно вести разными способами, и при этом будут получаться разные функции $\varphi(x)$в правой части уравнения. Уравнение $f(x)=0$эквивалентно уравнению $x = x + \lambda(x)f(x)$при любой функции $\lambda(x) \neq 0$. Таким образом, можно взять $\varphi(x) = x + \lambda(x)f(x)$и при этом выбрать функцию (или постоянную) $\lambda \neq 0$так, чтобы функция $\varphi(x)$ удовлетворяла тем свойствам, которые понадобятся нам для обеспечения нахождения корня уравнения.

Для нахождения корня уравнения $x=\varphi(x)$ выберем какое-либо начальное приближение x_0 (расположенное, по возможности, близко к корню x^*). Далее будем вычислять последующие приближения $x_1, x_2, \ldots, x_i, x_{i+1}, \ldots$ по формулам $x_1 = \varphi(x_0); x_2 = \varphi(x_1); \ldots; x_{i+1} = \varphi(x_i); \ldots$,то есть используя каждое вычисленное

приближение к корню в качестве аргумента функции $\varphi(x)$ в очередном вычислении. Такие вычисления по одной и той же формуле $x_{i+1} = \varphi(x_i)$, когда полученное на предыдущем шаге значение используется на последующем шаге, называются итерациями. Итерациями называют часто и сами значения x_i, полученные в этом процессе (то есть, в нашем случае, последовательные приближения к корню).

Заметим: тот факт, что x^* - корень уравнения $x = \varphi(x)$, означает, что x^* есть абсцисса точки пересечения графика $y = \varphi(x)$ с прямой $y = x$. Если же при каком-либо x_0 вычислено значение $x_1 = \varphi(x_0)$ и взято в качестве нового аргумента функции, то это означает, что через точку графика $(x_0, \varphi(x_0))$ проводится горизонталь до прямой $y = x$, а оттуда опускается перпендикуляр на ось Ox. Там и будет находиться новый аргумент x_1.

Если функция $\varphi(x)$ имеет производную в некоторой окрестности E корня x^* уравнения $x = \varphi(x)$, причём $|\varphi'(x)| \le \gamma < 1$ при $x \in E$, то последовательность итераций $x_{i+1} = \varphi(x_i)$, полученных при $i = 1,2,3,\ldots$, начиная с $x_0 \in E$, сходится к корню x^*.

При этом скорость сходимости задаётся неравенствами

$$|x_i - x^*| \le \gamma^i |x_0 - x^*|,\ i = 1,2,3,\ldots,$$

$$|x_{i+1} - x_i| \le 4\delta\gamma^i,$$

Где 2δ - длина окрестности E, а точность i-го приближения - оценкой $|x_i - x^*| \le 2\delta\gamma^i$.

Поскольку привести уравнение $f(x) = 0$ к виду $x = \varphi(x)$ можно, выбирая $\varphi(x)$ в виде $\varphi(x) = x + \lambda(x) f(x)$, где $\lambda(x) \ne 0$ - произвольная функция, то при различных способах выбора $\lambda(x)$ получаются разные модификации метода итераций, которые имеют отличающиеся свойства: разную скорость сходимости (но не меньшую той, что гарантирована теоремой) и разную потребность в вычислении значений функции f или φ, а также их производных.

Постановка задачи

Дано: функция $f(x,y)$, вещественные c, d и целое m (m>0). Для каждого значения $y_i = c + ih \quad (i = 0,1,\ldots,m)$, где $h = (d-c)/m$, найти корень x_i уравнения $F(x) \equiv f(x,y_i) = 0$. Если уравнение имеет более одного корня, то найти положительный, а среди корней одного знака найти наименьший по модулю (но отличный от нуля) корень.

Каждый корень найти с точностью $\varepsilon = 0.001$, пользуясь одним из приближенных методов.

Функция $f(x,y)$, значения c, d и m и приближенный метод нахождения корня определяются вариантом задания.

Значения c, d и m должны либо вводиться с клавиатуры в процессе выполнения программы, либо задаваться константами в программе. В качестве результатов необходимо вывести на экран значения y_i, соответствующие им значения x_i и значения $f(x_i, y_i)$. При выводе значения должны быть выровненными и содержать 6 знаков после запятой. В качестве заполнителя использовать символ '.'.

Для решения задачи следует исследовать заданную функцию; найти отрезок, в котором находится требуемый корень уравнения при любом значении y_i; проверить применимость указанного численного метода.

Решение задачи должно быть оформлено в виде подпрограммы, параметрами-аргументами которой являются c, d, m, ε и $f(x,y)$.

Содержание отчета

1. Постановка задачи (конкретный вариант).
2. Аналитическое исследование функции и проверка применимости указанного численного метода.
3. Спецификации подпрограмм, необходимых для решения задачи.
4. Результаты решения.
5. Текст программы на электронном носителе.

Варианты задания

Для определения варианта задания следует по номеру k (порядковый номер студента в списке), выбрать номер метода приближенного решения уравнения равным остатку от деления k на 4 и номер правила для вычисления функции $f(x,y)$, равным остатку от деления k на 9.

Метод приближенного решения уравнения $F(x)=0$.

0. Метод хорд (секущих).
1. Метод касательных (Ньютона).
2. Комбинированный метод (хорд и касательных).
3. Метод простых итераций.

Функция $f(x,y)$, значения *c*, *d* и *m.*

№№	$f(x,y)$	c	d	m
0	$\frac{1}{1+x^2}-yx$	1	2	10
1	$\mathrm{tg}^2 x-yx$	1	2	20
2	$e^{-x}-yx$	1	2	15
3	$\frac{1}{1+x^4}-yx^2$	1	2	30
4	$\ln(2+x)-yx^3$	5	6	10
5	$x^3-10-y\sqrt{x-2}$	1	2	15
6	$\sqrt{1-x^2}-y^2x^5$	$\sqrt{2}$	$\sqrt{3}$	20
7	$1-x^2-ye^x$	$\frac{1}{3}$	$\frac{1}{2}$	12
8	$\sin 2x-yx^2$	1	10	30

Литература

1. Березин И. С., Жидков Н. П. Методы вычислений. Т. 1. – М.: Наука, 1966. – с. 127 – 136.
2. Фихтенгольц Г. М. Курс дифференциального и интегрального исчисления. М.: Физматлит, 2001. т.1 - 616с.

Задание по учебной практике №3.
Работа со строками. Связное хранение строк.

Цепочки

Наиболее простой способ объединить или связать некоторое множество элементов — это «вытянуть их в линию», организовав цепочку. В этом случае с каждым звеном нужно сопоставить лишь одну-единственную ссылку, указывающую на следующее звено. Каждое звено цепочки в этом случае представляет собой запись, состоящую из двух компонент, а именно самого данного и ссылки на следующий элемент (назовем этот компонент next).

Графически связи в цепочках удобно изображать с помощью стрелок. Если звено не связано ни с каким другим, то в ссылочном поле записывают значение, не указывающее ни на какое звено. Такая ссылка обозначается специальным именем – nil и является признаком последнего звена цепочки.

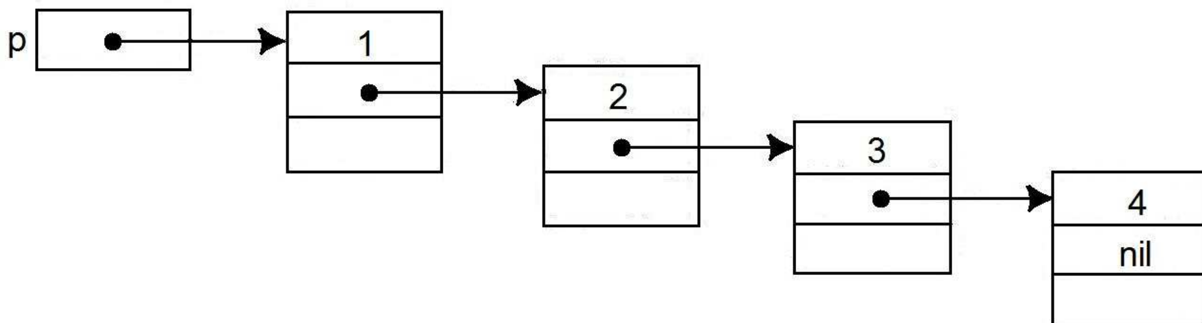

Рис. 1. Пример цепочки

На Рис. 1 представлена цепочка вершин, ссылка на первое звено которой присвоена переменной p. Вероятно, самая простая операция, которую можно проделать с такой цепочкой, — включить в ее начало некоторый элемент. Сначала элемент типа запись размещается в памяти, и ссылка на него присваивается некоторой вспомогательной переменной q. После этого ссылкам присваиваются новые значения, и операция на этом заканчивается.

Описанная таким образом операция включения элемента в голову цепочки сразу же объясняет, как можно вообще формировать любую цепочку: начать с пустой цепочки и последовательно добавлять в ее начало элементы.

Это самый простой способ построения цепочки, однако при нем порядок элементов в цепочке обратен порядку их включения. В некоторых случаях это нежела-

тельно, и поэтому новые элементы необходимо добавлять не в голову, а в конец цепочки. Конец можно легко найти, просматривая всю цепочку, но это слишком «наивное» решение, требующее определенных затрат, от которых просто избавиться. Достаточно ввести вторую ссылку q, указывающую на последнее звено цепочки. Недостаток этого метода в том, что первый из включаемых элементов нужно обрабатывать иначе, чем остальные.

Явное использование ссылок намного упрощает некоторые операции, которые иначе были бы излишне запутанными. Среди элементарных операций над цепочками есть включение и исключение элементов (выборочное изменение цепочки) и, конечно же, просмотр цепочки. Начнем с разбора операции *включения в* цепочку.

Предположим, после звена цепочки, на которое указывает ссылка p, нужно включить звено, заданное ссылкой q (переменной).

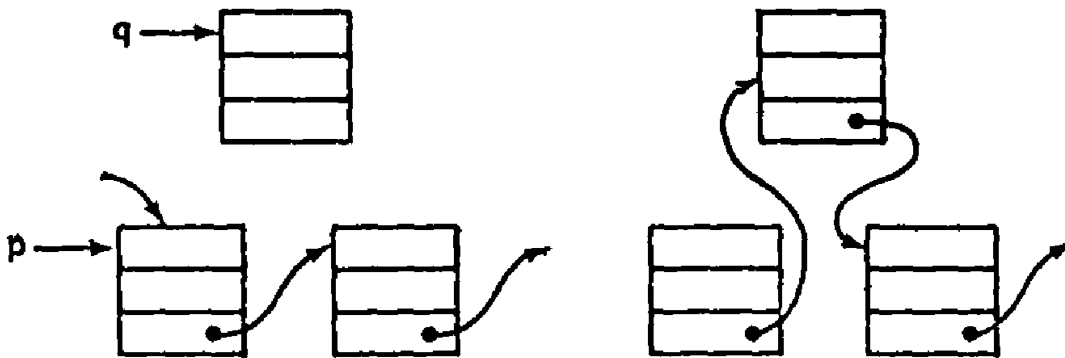

Рис. 2. Включение в цепочку после звена, на которое указывает ссылка p.

Такое включение выполняется двумя операторами:

1 Полю next звена цепочки, на который указывает ссылка q, присваивается значение поля next звена цепочки, на который указывает ссылка p;

2 Полю next звена цепочки, на который указывает ссылка p, присваивается значение переменной q.

Если требуется включение не после, а *перед* звеном, на который указывает ссылка p, то кажется, что однонаправленная цепочка связей должна затруднить работу, поскольку нет «хода» к звеньям, предшествующим данному. Однако эта проблема решается довольно просто, а именно: новая компонента на самом деле включается *после* звена, на которое указывает ссылка p, но затем новая звено и звено, на которое указывает ссылка p «меняются» значениями.

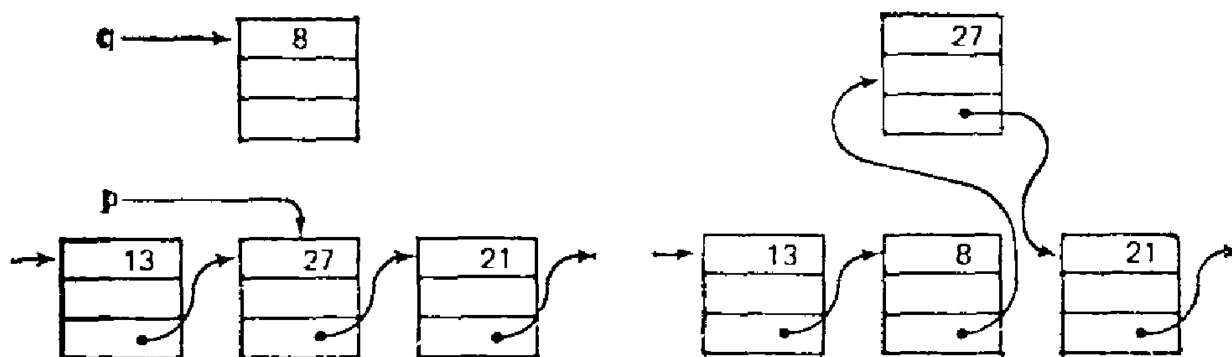

Рис. 3. Включение в список перед элементом, на который указывает ссылка p.

Теперь рассмотрим процесс *исключения из цепочки.* Исключение звена, следующего за звеном, на которое указывает ссылка p, очевидно. Для этого достаточно изменить значение поля next звена, на которое указывает ссылка p, на значение поля next звена, содержащего исключаемый элемент.

Труднее исключить само звено, содержащее указанный элемент (а не следующее за ним), поскольку мы сталкиваемся с той же проблемой, что и при включении. Ведь проход к звену, предшествующему указанному, невозможен. Однако теперь уже довольно очевидно простое решение: исключается последующее звено, а перед этим содержащееся в нем значение «передвигается» вперед. Так можно поступать, когда у звена, на которое указывает ссылка p, есть последующее, т. е. если это не последнее звено.

Разберем теперь следующую операцию — *проход по цепочке.* Предположим, что с каждым из звеньев цепочки нужно выполнить операцию P(x); первое звено цепочки – это звено, на который указывает ссылка p. Эту задачу можно выполнить следующим образом:

ПОКА *цепочка, обозначенная через p, непуста ВЫПОЛНЯТЬ*
        P;
        переход к следующему элементу
ВСЕ

Переход к следующему элементу состоит в присваивании ссылке p значения поля next звена, на которое она указывает.

Из определения оператора цикла с предусловием и цепочной структуры следует, что P будет выполнено для всех звеньев цепочки, и ни для каких других.

Одна из наиболее часто употребляемых операций - *поиск в цепочке* данного с заданным значением x. В отличие от массивов поиск в этом случае должен проходить строго последовательно. Он заканчивается либо при обнаружении данного, либо при достижении конца цепочки. Такие условия приводят к тому, что конъюнкция включает два отношения: поле next звена, на которое указывает ссылка p, не равно nil (пустому

значению, которое идентифицирует последний элемент цепочки) и поле данного этого звена не равно искомому значению.

Если наряду с данным в каждом звене цепочки хранятся ссылки не только на следующий, но и на предыдущий элемент, такая цепочка называется двунаправленной. Операции включения – исключения в такой цепочке выполняются проще, чем в однонаправленной, однако надо помнить о том, что эти операции влекут за собой изменения ссылочных полей у обоих «соседей».

Если значение ссылочного поля последнего звена цепочки равно значению ссылки на ее первое звено, такая цепочка называется циклической. В циклической двунаправленной цепочке, кроме того, значение ссылочного поля, указывающего на предыдущее звено, первого звена цепочки равно значению ссылки на ее последнее звено.

Постановка задачи

Дан некоторый текст, где

<текст> ::= <предложение>**.**

<предложение> ::= <слово> **|** <предложение><разделитель><слово>

<разделитель> ::=

пробел{пробел}|символ табуляции|символ конца строки|.|,|;|:|!|?|-|‘|“

Предполагается, что слово содержит не более 20 букв. Количество слов в предложении не ограничивается.

Требуется:

1. Прочитать исходный текст из файла и поместить его в памяти, используя заданный вариантом способ связного хранения. Каждое звено всех перечисленных ниже внутренних представлений текста размещается в памяти динамически и содержит строку, изображающую слово, а также ссылку на следующее звено или две ссылки – на следующее и предыдущее звенья в зависимости от варианта;
2. Текст в его внутреннем представлении переработать по заданному вариантом правилу;
3. Переработанный текст вывести на экран, разделив слова пробелами (по одному между соседними словами), предусмотрев, чтобы при печати в словах не было переносов и текст был выровнен по правому краю экрана.

Для решения задачи рекомендуется использовать следующие подпрограммы:

1. Подпрограмму, добавляющую слово в цепочку;
2. Подпрограмму, возвращающую очередное слово текста;

3. Подпрограмму для построения цепочки слов;
4. Подпрограмму для переработки текста по правилу, определяемому вариантом;
5. Подпрограмму для форматированного вывода текста на экран.

Содержание отчета

1. Постановка задачи (конкретный вариант).
2. Спецификации подпрограмм, необходимых для решения задачи.
3. Результаты решения.
4. Текст программы на электронном носителе.

Варианты задания.

Для определения варианта задания следует по номеру k (порядковый номер студента в списке), выбрать номер вида внутреннего представления текста равным остатку от деления k на 4 и номер правила переработки текста, равным остатку от деления k на 12.

Вид внутреннего представления текста.

0. Однонаправленная цепочка;
1. Однонаправленная циклическая цепочка;
2. Двунаправленная цепочка;
3. Двунаправленная циклическая цепочка.

Правила переработки текста

0. Из текста удалить все последующие вхождения его первого слова, содержащего две одинаковые буквы;
1. Между любыми двумя словами исходного текста вставить слово “or”. Если в текст первоначально слово “or” входило, удалить все его вхождения;
2. Удалить из текста все слова, в которых есть одинаковые буквы;
3. В каждой очередной паре слов поменять слова местами, причем второе слово перевернуть;
4. Из каждого слова текста, содержащего более четырех букв, удалить три буквы, непосредственно предшествующие последней букве слова. Слова, содержащие менее четырех букв, из текста удалить;
5. Каждое вхождение в текст его первого однобуквенного слова заменить словом “yes”, все другие однобуквенные слова удалить из текста;

6. Если последнее двухбуквенное слово текста входит в него несколько раз, то оставить только последнее вхождение;
7. В каждой очередной тройке слов первое слово продублировать после третьего.
8. Если первое однобуквенное слово входит в текст не менее трех раз, то удалить часть текста, расположенную между вторым и третьим вхождением этого слова;
9. В тексте оставить только первое вхождение каждого из слов, начинающихся с одинаковой буквы. Остальные вхождения удалить;
10. В тексте объединить слова попарно;
11. Вставить после каждого слова его перевернутую копию. Слова – палиндромы удалить.

Литература

1 Вирт Н. Алгоритмы и структуры данных. -Москва «Мир», 1989. -360с.

Задание по учебной практике № 4. Реализация методов сортировок

Алгоритмы сортировки

Алгоритм сортировки — это алгоритм для упорядочения элементов в массиве. В случае, когда элемент массива имеет несколько полей, поле, служащее критерием порядка, называется ключом сортировки. На практике в качестве ключа часто выступает число, а в остальных полях хранятся какие-либо данные, никак не влияющие на работу алгоритма.

Алгоритмы сортировки оцениваются по скорости выполнения и эффективности использования памяти:

Время - основной параметр, характеризующий быстродействие алгоритма. Он называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах размера списка (n). Для типичного алгоритма хорошее поведение — это O(n log n), и плохое поведение — это $O(n^2)$. Идеальное поведение для упорядочения —O(n). Алгоритмы сортировки, использующие только абстрактную операцию сравнения ключей всегда нуждаются по меньшей мере в O(n log n), сравнениях в среднем. Тем не менее, существует алгоритм сортировки Хана с вычислительной сложностью O(n log n), использующий тот факт, что пространство ключей ограничено (он чрезвычайно сложен, а за О-обозначением скрывается весьма большой коэффициент, что делает невозможным его применение в повседневной практике).

Память - ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.

Алгоритмы классифицируются по следующим признакам:

Устойчивость— устойчивая сортировка не меняет взаимного расположения равных элементов.

Естественность поведения — эффективность метода при обработке уже упорядоченных, или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.

Использование операции сравнения. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях. Мини-

мальная трудоемкость худшего случая для этих алгоритмов составляет O(n log n), но они отличаются гибкостью применения. Для специальных случаев (типов данных) существуют более эффективные алгоритмы.

потребность в дополнительной памяти или её отсутствии

потребность в знаниях о структуре данных, выходящих за рамки операции сравнения, или отсутствии таковой.

Сортировка бинарными вставками

Алгоритм сортировки вставкой подразумевает виртуальное разделение массива на две части - слева отсортированную и справа ещё неотсортированную. Каждый первый элемент правого подмассива вставляется в левый так, чтобы его упорядоченность не нарушилась. То есть алгоритм вставки действует, например, так:

массив 4 7 1 10 3

подразумеваем, что самое начало отсортировано

4 | 7 1 10 3

вставляем 7 после 4

4 7 | 1 10 3

1 надо вставить перед 4

1 4 7 | 10 3

10 - после 7

1 4 7 10 | 3

3 - между 1 и 3

1 3 4 7 10|. Всё

Таким образом, задача сортировки сведена к поиску номера элемента массива, который меньше заданного числа, но следующий за ним элемент уже больше заданного числа. Применяется метод бинарного поиска. Этот метод применим только к отсортированному массиву.

Принцип следующий. Рассматривается сначала целый массив, допустим, [1 4 7 10] (вставка тройки). Он разбивается на две половинки, и граничный элемент сравнивается с заданным числом; если число превосходит границу, оно должно быть вставлено в правую половинку - так что рассматриваем её, иначе берём левую половинку. $3<4 \Rightarrow$ [1 4] 7 10. Далее повторяем алгоритм - рассматриваемую часть уменьшаем вдвое и выбираем место, куда вставить требуемое число. $3>1 \Rightarrow$ 1 [4] 7 10 ; $3<4 \Rightarrow$ 1 [] 4 7 10. Вставляем - 1 [3] 4 7 10.

Реализация в приложении. Подпрограмма sort принимает два параметра - сортируемый массив и его размер. Ничего не возвращает, просто модифицирует заданный массив.

Сортировка двухпутевыми вставками

Впервые метод двухпутевых вставок был предложен в начале 50-х годов как метод, позволяющий сократить число необходимых перемещений (по сравнению с простыми вставками). Число пересылок можно сократить примерно в 2 раза до N/8, если допустить сдвиги элементов не только вправо, но и влево. Для выходного файла резервируется место в памяти, равное 2N+1 ,где N – число элементов в исходном файле. Первый элемент пересылается в середину выходного файла. В дальнейшем элементы выходного файла сдвигаются вправо или влево в зависимости от того, в какую сторону нужно сдвигать меньше элементов, то есть туда, где удобнее. Таким образом, удается сократить примерно половину времени работы за счет некоторого усложнения программы. Но этот метод можно применять, используя памяти не больше, чем требуется для N записей. Рассмотрим работу алгоритма на следующем примере.

503 87 512 61 908 170 897 275

Процесс сортировки выглядит следующим образом:

Первый элемент (503)помещаем в середину области вывода.

Устанавливаем положение следующего элемента (87) Сравниваем его с левой границей (87<503). 087 - левая граница в выходном массиве

87 503

Сравниваем следующую запись (512) с правой границей. Эта запись становится новой правой границей.

Следующую запись (61) сравниваем с левой границей. 61 становится левой границей в выходном массиве

61 87 503 512

Сравниваем запись №5 (908) с правой границей Запись №5 становится новой правой границей.

61 87 503 512 908

Сравниваем запись №6 (170) с левой границей. Сравниваем запись №6 с правой границей. Находим место для записи №6 - позиция 3. Производим сдвиг.

61 87 170 503 512 908

Дальше продолжаем аналогично до конца ряда. В результате получаем последовательность:

61 87 170 275 426 503 512 653 897 908.

Сортировка методом Шелла

Сортировка Шелла (англ. Shell sort) — алгоритм сортировки, идея которого состоит в сравнении элементов, стоящих не только рядом, но и на расстоянии друг от друга. Иными словами - сортировка вставками с предварительными "грубыми" проходами.

При сортировке Шелла сначала сравниваются и сортируются между собой ключи, отстоящие один от другого на некотором расстоянии d. После этого процедура повторяется для некоторых меньших значений d, а завершается сортировка Шелла упорядочиванием элементов при d = 1 (то есть, обычной сортировкой вставками. Эффективность сортировки Шелла в определённых случаях обеспечивается тем, что элементы «быстрее» встают на свои места (в простых методах сортировки вставками или пузырьком (но она не предпочтительна, так как все равно остается медленной) каждая перестановка двух элементов уменьшает количество инверсий в списке максимум на 1, при сортировке Шелла же это число может быть больше).

Невзирая на то, что сортировка Шелла во многих случаях медленнее, чем быстрая сортировка, она имеет ряд преимуществ:

отсутствие потребности в памяти под стек

отсутствие деградации при неудачных наборах данных – быстрая сортировка легко деградирует до $O(n^2)$, что хуже, чем худшее гарантированное время для сортировки Шелла

Часто оказывается, что сортировка Шелла есть самый лучший способ сортировки до, примерно, 1000 элементов.

Сортировка вставками в список.

Среди общих способов улучшения алгоритма один из самых важных основывается на тщательном анализе структур данных, поскольку реорганизация структур данных, позволяющая избежать ненужных операций, часто дает существенный эффект. Рассмотрим, как она применяется к алгоритму простых вставок. Какова наиболее подходящая структура данных для данного алгоритма? Сортировка методом простых вставок состоит из двух основных операций:

1. просмотра упорядоченного массива для нахождения наибольшего ключа, меньшего или равного данному ключу;

2. вставки новой записи в определенное место упорядоченного массива.

Массив — это, очевидно, линейный список, и алгоритм обрабатывает его, используя последовательное распределение; поэтому для выполнения каждой операции вставки необходимо переместить примерно половину записей. С другой стороны, нам известно, что для вставок идеально подходит хранение данных в памяти в виде связного списка, так как при этом требуется изменить лишь несколько связей; другая операция — последовательный просмотр — при использовании связного списка почти так же проста, как и при последовательном размещении данных. Поскольку списки всегда просматриваются в одном и том же направлении, достаточно иметь списки с однонаправленной связью. Таким образом, приходим к выводу, что "правильная" структура данных для метода простых вставок — линейные списки с однонаправленными связями. В итоге, приходим к следующему алгоритму *(метод вставки в список).* Предполагается, что записи $R_1,...,R_N$ содержат ключи $K_1,..., K_N$ и поля связи $L_1,..., L_N,$ в которых могут храниться числа от 0 до N. Алгоритм устанавливает поля связи так, что записи оказываются связанными в порядке возрастания.

Быстрая сортировка.

Быстрая сортировка использует стратегию «разделяй и властвуй». Шаги алгоритма таковы:

Выбираем в массиве некоторый элемент, который будем называть опорным элементом. С точки зрения корректности алгоритма выбор опорного элемента безразличен. С точки зрения повышения эффективности алгоритма выбираться должна медиана, но без дополнительных сведений о сортируемых данных её обычно невозможно получить. Известные стратегии: выбирать постоянно один и тот же элемент, например, средний или последний по положению; выбирать элемент со случайно выбранным индексом.

Операция разделения массива: реорганизуем массив таким образом, чтобы все элементы, меньшие или равные опорному элементу, оказались слева от него, а все элементы, большие опорного - справа от него. Обычный алгоритм операции:

два индекса - l и r, приравниваются к минимальному и максимальному индексу разделяемого массива соответственно;

вычисляется опорный элемент m;

индекс l последовательно увеличивается до m или до тех пор, пока l-й элемент не превысит опорный;

индекс r последовательно уменьшается до m или до тех пор, пока r-й элемент не окажется меньше опорного;

если r = l — найдена середина массива — операция разделения закончена, оба индекса указывают на опорный элемент;

если l < r — найденную пару элементов нужно обменять местами и продолжить операцию разделения с тех значений l и r, которые были достигнуты. Следует учесть, что если какая-либо граница (l или r) дошла до опорного элемента, то при обмене значение m изменяется на r или l соответственно.

Рекурсивно упорядочиваем подмассивы, лежащие слева и справа от опорного элемента.

Базой рекурсии являются наборы, состоящие из одного или двух элементов. Первый возвращается в исходном виде, во втором, при необходимости, сортировка сводится к перестановке двух элементов. Все такие отрезки уже упорядочены в процессе разделения.

Поскольку в каждой итерации (на каждом следующем уровне рекурсии) длина обрабатываемого отрезка массива уменьшается, по меньшей мере, на единицу, терминальная ветвь рекурсии будет достигнута всегда и обработка гарантированно завершится.

Интересно, что Хоар разработал этот метод применительно к машинному переводу: дело в том, что в то время словарь хранился на магнитной ленте, и если упорядочить все слова в тексте, их переводы можно получить за один прогон ленты.

Сортировка слиянием

Сортировка слиянием (англ. merge sort) — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке. Эта сортировка — хороший пример использования принципа «разделяй и властвуй». Сначала задача разбивается на несколько подзадач меньшего размера. Затем эти задачи решаются с помощью рекурсивного вызова или непосредственно, если их размер достаточно мал. Наконец, их решения комбинируются, и получается решение исходной задачи.

Для решения задачи сортировки эти три этапа выглядят так:

1) Сортируемый массив разбивается на две половины примерно одинакового размера;

2) Каждая из получившихся половин сортируется отдельно, например - тем же самым алгоритмом;

3) Два упорядоченных массива половинного размера соединяются в один.

Рекурсивное разбиение задачи на меньшие происходит до тех пор, пока размер массива не достигнет единицы (любой массив длины 1 можно считать упорядоченным).

Нетривиальным этапом является соединение двух упорядоченных массивов в один. Основную идею слияния двух отсортированных массивов можно объяснить на следующем примере. Пусть мы имеем две стопки карт, лежащих рубашками вниз так, что в любой момент мы видим верхнюю карту в каждой из этих стопок. Пусть также, карты в каждой из этих стопок идут сверху вниз в неубывающем порядке. Как сделать из этих стопок одну? На каждом шаге мы берём меньшую из двух верхних карт и кладём её (рубашкой вверх) в результирующую стопку. Когда одна из оставшихся стопок становится пустой, мы добавляем все оставшиеся карты второй стопки к результирующей стопке.

Алгоритм был изобретён Джоном фон Нейманом в 1945 году.

Время работы алгоритма порядка O(n log n) при отсутствии деградации на неудачных случаях, которая есть больное место быстрой сортировки (тоже алгоритм порядка O(n log n), но только для лучшего случая). Расход памяти выше, чем для быстрой сортировки, при намного более благоприятном паттерне выделения памяти - возможно выделение одного региона памяти с самого начала и отсутствие выделения при дальнейшем исполнении.

Сортировка распределением (поразрядная сортировка)

Это метод, совершенно отличный от всех предыдущих схем сортировки. В нем используется *двоичное представление* ключей. Вместо того чтобы сравнивать между собой два ключа, в этом методе проверяется, равны ли 0 или 1 отдельные разряды ключа. В других отношениях он обладает характеристиками обменной сортировки и на самом деле очень напоминает быструю сортировку. Так как метод использует разряды ключа, представленного в двоичной системе счисления, его называют поразрядной сортировкой. В общих чертах этот алгоритм можно описать следующим образом.

1. Последовательность сортируется по *старшему значащему двоичному разряду* так, чтобы все ключи, начинающиеся с 0, оказались перед всеми ключами, начинающимися с 1. Для этого необходимо найти крайний слева ключ *Ki,* начинающийся с 1, и крайний справа ключ *Kj,* начинающийся с 0, после чего

R_i и Rj поменяются местами, и процесс будет повторяться, пока не получится $i > j$.

2. Пусть F_0 — множество элементов, начинающихся с 0, а F_i — множество всех остальных элементов. Будем применять к F_0 поразрядную сортировку (начав теперь со *второго* бита слева, а не со старшего) до тех пор, пока множество F_0 полностью не рассортируется. Затем проделаем то же самое с F_1.

Как и при быстрой сортировке, для хранения "информации о границах" подмассивов, ожидающих сортировки, можно воспользоваться стеком. Вместо того чтобы сортировать, в первую очередь, наименьший из подмассивов, удобно просто продвигаться слева направо, и размер стека в этом случае никогда не превзойдет числа разрядов в двоичном представлении сортируемых ключей. Левую границу при этом можно не запоминать в стеке: она всегда будет задана неявно.

Постановка задачи.

Дан информационный массив из *m* (*m>0*) записей об итогах сдачи сессии студентами ИМЭМ следующей структуры:

<table>
<tr><td>1</td><td>2</td><td>3</td><td colspan="4">4</td></tr>
<tr><td rowspan="3">Курс</td><td rowspan="3">Группа</td><td rowspan="3">Фамилия И. О.</td><td colspan="4">Количество оценок</td></tr>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>Отлично</td><td>Хорошо</td><td>Удовлетворительно</td><td>Неудовлетворительно</td></tr>
</table>

Номер курса задается целым числом из диапазона 1÷5.

Название группы - строка не более чем из 5 символов.

Фамилия и инициалы - строка не более чем из 20 символов.

Количество оценок – целое неотрицательное число.

Требуется упорядочить этот информационный массив, пользуясь одним из методов сортировок.

Исходный информационный массив вводится из файла.

Упорядочение массива выполняется по составному ключу. Массив упорядочивается по неубыванию значений первого поля составного ключа. Если в массиве имеется несколько записей с одинаковыми значениями первого поля составного ключа, то эти записи упорядочиваются по неубыванию значений второго поля составного ключа и т. д. Для символьных полей упорядочение лексикографическое.

В программе предусмотреть вывод на экран исходного массива, упорядоченного массива и характеристик сортировок: количество выполненных сравнений и количество перестановок элементов.

Число m выбирается произвольно, методы сортировок и компоненты составного ключа определяются как остаток от деления k (номера студента в списке) на 7 и 12 соответственно.

Содержание отчета

1. Постановка задачи (конкретный вариант).
2. Спецификации подпрограмм, необходимых для решения задачи.
3. Результаты решения.
4. Текст программы на электронном носителе.

Варианты задания.

Методы сортировок

0. Сортировка бинарными вставками.
1. Сортировка двухпутевыми вставками.
2. Сортировка методом Шелла.
3. Сортировка вставками в список.
4. «Быстрая сортировка» Хоара.
5. Сортировка слиянием.
6. Сортировка распределением (поразрядная сортировка).

Значение *m* и составной ключ.

№ №	m	Составной ключ
0	10	1, 3
1	15	2, 3
2	20	1, 2, 3
3	20	2, 1, 3
4	25	4.1, 4.2, 3
5	15	1, 4.1, 4.2
6	10	4.1, 4.2, 4.3
7	10	1, 4.1, 3
8	20	4.4, 1, 3
9	15	4.4, 2, 3
10	25	1, 4.4, 3
11	20	1, 2, 4.4

Методические указания для выполнения задания №4

Решение задачи должно использовать приемы обобщенного программирования, т. е. тип сортируемого массива должен передаваться в подпрограмму как параметр. Поэтому язык программирования для решения этой задачи должен поддерживать возможность обобщенного описания подпрограмм. Кроме того, вышеуказанное требование диктует введение операций сравнения для типа сортируемого массива (в задании №4 сортируемый массив – это массив записей). Значит, в языке программирования должна поддерживаться перегрузка операций. Оба механизма поддерживаются, в частности, в языке программирования C++.

Литература

1. Д. Кнут. Искусство программирования. т. 3 –Москва, Вильямс, 2000. -822с.

Задание по учебной практике № 5. Программирование лексического анализатора

Лексический анализ

Основная задача лексического анализа - разбить входной текст, состоящий из последовательности одиночных символов, на последовательность слов, или лексем, т.е. выделить эти слова из непрерывной последовательности символов. Все символы входной последовательности с этой точки зрения разделяются на символы, принадлежащие каким-либо лексемам, и символы, разделяющие лексемы (разделители). В некоторых случаях между лексемами может и не быть разделителей. С другой стороны, в некоторых языках лексемы могут содержать незначащие символы (например, символ пробела в Фортране). В Си разделительное значение символов-разделителей может блокироваться («\» в конце строки внутри "...").

Обычно все лексемы делятся на классы. Примерами таких классов являются числа (целые, восьмеричные, шестнадцатиричные, действительные и т.д.), идентификаторы, строки. Отдельно выделяются ключевые слова и символы пунктуации (иногда их называют символы-ограничители). Как правило, ключевые слова - это некоторое конечное подмножество идентификаторов. В некоторых языках (например, ПЛ/1) смысл лексемы может зависеть от ее контекста и невозможно провести лексический анализ в отрыве от синтаксического.

С точки зрения дальнейших фаз анализа лексический анализатор выдает информацию двух сортов: для синтаксического анализатора, работающего вслед за лексическим, существенна информация о последовательности классов лексем, ограничителей и ключевых слов, а для контекстного анализа, работающего вслед за синтаксическим, важна информация о конкретных значениях отдельных лексем (идентификаторов, чисел и т.д.).

Таким образом, общая схема работы лексического анализатора такова. Сначала выделяется отдельная лексема (возможно, используя символы-разделители). Ключевые слова распознаются либо явным выделением непосредственно из текста, либо сначала выделяется идентификатор, а затем делается проверка на принадлежность его множеству ключевых слов.

Если выделенная лексема является ограничителем, то он (точнее, некоторый его признак) выдается как результат лексического анализа. Если выделенная лексема является ключевым словом, то выдается признак соответствующего ключевого слова. Если

выделенная лексема является идентификатором - выдается признак идентификатора, а сам идентификатор сохраняется отдельно. Наконец, если выделенная лексема принадлежит какому-либо из других классов лексем (например, лексема представляет собой число, строку и т.д.), то выдается признак соответствующего класса, а значение лексемы сохраняется отдельно.

Лексический анализатор может быть как самостоятельной фазой трансляции, так и подпрограммой, работающей по принципу «дай лексему». В первом случае (Рис. 4) выходом анализатора является файл лексем, во втором (Рис. 5) лексема выдается при каждом обращении к анализатору (при этом, как правило, признак класса лексемы возвращается как результат функции «лексический анализатор», а значение лексемы передается через глобальную переменную). С точки зрения обработки значений лексем, анализатор может либо просто выдавать значение каждой лексемы, и в этом случае построение таблиц объектов (идентификаторов, строк, чисел и т.д.) переносится на более поздние фазы, либо он может самостоятельно строить таблицы объектов. В этом случае в качестве значения лексемы выдается указатель на вход в соответствующую таблицу.

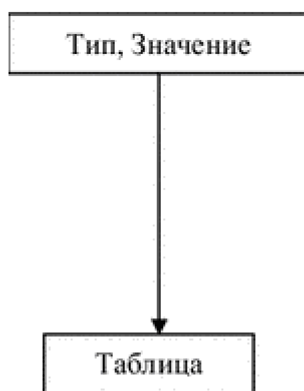

Рис. 4. Лексический анализатор как самостоятельная фаза трансляции

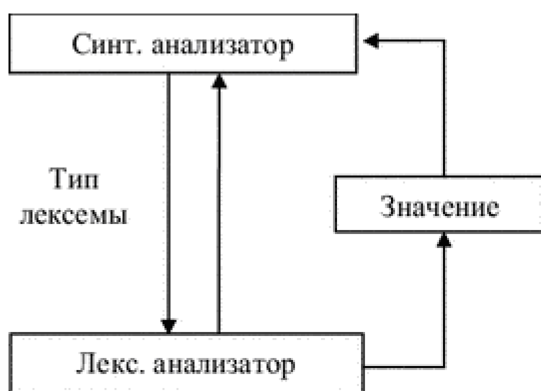

Рис. 5. Лексический анализатор как подпрограмма, работающая по принципу «дай лексему».

Работа лексического анализатора задается некоторым конечным автоматом. Кроме того, для задания лексического анализатора часто используется либо регулярное выражение, либо праволинейная грамматика. Все три формализма (конечных автоматов, регулярных выражений и праволинейных грамматик) имеют одинаковую выразительную мощность. В частности, по регулярному выражению или праволинейной грамматике можно сконструировать конечный автомат, распознающий тот же язык.

Постановка задачи.

Предположим, что некоторый язык высокого уровня содержит некоторые из следующих видов элементарных конструкций - лексем :

0. целые константы, положительные или нулевые - примеры: **423 0 8492**;
1. вещественные константы, состоящие из непустых целой и дробной части, разделенных точкой, - примеры: **24.37 0.64**;
2. знаки одноместных операций: **+ - * / = ( ) < >**;
3. знаки двуместных операций: **<= >= <> := ****;
4. знаки препинания: **; : , .**;
5. идентификаторы, - примеры: **X Y1 T02A PAPA** (длина идентификаторов не ограничивается; буквы могут использоваться только заглавные латинские);
6. разделители: серии из одного или нескольких пробелов, знак табуляции или признак "конец строки";
7. комментарии - последовательность символов между (* и *), примеры: (* это комментарий *);
8. строки - последовательность символов, заключенная в одинарные кавычки, примеры : 'СТРОКА' 'ABC*DE'.

Требуется построить функцию, способную выделить в анализируемом тексте все входящие в него лексемы языка и выводящую в результирующий файл саму лексему и ее тип.

Исходный текст для лексического анализа должен находиться во внешнем файле. Состав лексем конкретного языка определяется вариантом задания (остаток от деления k – номера студента в списке – на 9).

Для решения задачи следует:

1. Исследовать лексический состав заданного языка: определить классы лексем и литер, уточнить допустимое представление лексем.
2. Построить конечный автомат, реализующий лексический анализатор для заданного набора лексем.
3. Разработать алгоритм решения поставленной задачи на основе построенного конечного автомата.
4. Составить и отладить на функцию , которая для любой литеры с выдает ее класс.
5. Составить и отладить на функцию, которая для каждой лексемы возвращает строку с названием этой лексемы.
6. Составить и отладить программу решения поставленной задачи.

Требования к программе.

1. Программа должна правильно выделять из текста все имеющиеся в нем лексемы исходного языка, даже если в тексте имеются неверно построенные лексемы или встречаются недопустимые литеры.
2. Если в исходном тексте встречается недопустимый символ, функция должна выдать соответствующее сообщение, и продолжить обработку текста, предварительно удалив недопустимый символ. Если в исходном тексте встречается неверно построенная лексема (например, незакрытый комментарий или незакрытая строка), соответствующая функция должна квалифицировать эту лексему, как неправильную, после чего лексический анализ должен быть продолжен.
3. Реализацию диаграммы перехода необходимо выполнить программным путем с помощью структуры управления многозначный выбор.
4. В тексте программы предусмотреть поясняющие комментарии.

Содержание отчета

1. Постановка задачи (конкретный вариант).
2. Спецификации подпрограмм, необходимых для решения задачи.
3. Конечный автомат для конкретного варианта.
4. Результаты решения.
5. Текст программы на электронном носителе.

Варианты задания.

Набор лексем исходного языка.

№№	Лексемы
0	(1), (2), (5), (6), (7), (9)
1	(1), (3), (4), (6), (7), (8)
2	(3), (4), (6), (7), (8), (9)
3	(3), (4), (5), (6), (7), (9)
4	(1), (2), (3), (4), (6), (7)
5	(1), (2), (3), (5), (6), (7)
6	(1), (3), (6), (7), (8), (9)
7	(1), (2), (3), (7), (8), (9)
8	(1), (3), (5), (6), (7), (9)
9	(3), (4), (5), (6), (7), (8)
10	(3), (4), (5), (6), (7), (9)
11	(2), (3), (4), (5), (6), (7)
12	(1), (2), (3), (4), (5), (6)
13	(2), (4), (6), (7), (8), (9)
14	(2), (3), (5), (7), (8), (9)
15	(2), (3), (4), (5), (6), (7)
16	(2), (3), (4), (5), (6), (8)
17	(2), (3), (4), (5), (6), (9)

Методические указания для выполнения задания №5.

Решение задачи должно быть реализовано в виде подпрограммы, параметром - аргументом которой является текстовый файл с исходными данными, параметром результатом – результирующий текстовый файл, каждая строка которого состоит из самой лексемы и названия ее типа.

Для решения задачи удобно воспользоваться разбиением символов на следующие классы:

1. буква (**заглавные латинские буквы**)
2. цифра (**десятичные цифры от 1 до 9 включительно**)
3. ноль (**0**)
4. знак операции (/ + **-)**
5. *
6. =
7. (
8.)
9. <
10. >
11. :
12. .
13. пробел
14. разделитель (**один или несколько пробелов, знак табуляции или признак "конец строки"**)
15. '
16. конец текста
17. ошибочный символ

Литература

1. В.А.Серебряков, М. П. Галочкин. Основы конструирования компиляторов. –Едиториал Урсс. 2001. -224с.
2. Ахо А., Сети Р., Ульман Д. Компиляторы. Принципы, технологии, инструменты. –Вильямс. 2003. -768с.

Приложение. Некоторые средства языка C++, необходимые для выполнения заданий учебной практики.

Указатели на функции. Функции, как и элементы данных, имеют адреса. Адресом функции является адрес, с которого начинается машинный код функции, хранящийся в памяти. Обычно пользователи не извлекают никаких преимуществ из того, что знают этот адрес, но эти сведения могут оказаться полезными для программы. Например, можно создать функцию, которая принимает адрес другой функции в качестве аргумента. Это дает возможность первой функции найти вторую функцию и выполнить ее. Такой подход более громоздкий, чем прямое обращение ко второй функции со стороны первой, тем не менее, он позволяет осуществлять передачу адресов различных функций в различные моменты. Это означает, что первая функция может использовать различные функции в различные моменты.

Рассмотрим пример. Предположим, нужно спроектировать функцию estimate(), которая вычисляет время, необходимое для написания заданного числа строк программного кода. Часть программного кода функции estimate() будет одинаковой для всех пользователей, однако данная функция позволит каждому программисту воспользоваться собственным алгоритмом вычисления времени. Механизм реализации этого подхода состоит в передаче функции estimate() адреса функции, реализующей конкретный алгоритм, который пользователь желает использовать. Чтобы осуществить этот план, нужно выполнить следующее:

- Принять адрес функции.
- Объявить указатель на функцию.
- Использовать указатель на функцию для вызова этой функции.

Получение адреса функции несложно: достаточно воспользоваться именем функции без замыкающих скобок, т.е. если think() - функция, то think — адрес этой функции. Чтобы передать какую-либо функцию как аргумент, достаточно передать имя этой функции.

```
process(think); // передает функции process() адрес
     //функции think()
thought(think()); // передает функции // thought()
     // возвращаемое значение функции think ()
```

Вызов process() позволяет функции process() вызывать функцию think() из функции process(). Обращение к функции thought() сначала приводит к вызову функции think(), а затем к передаче значения, возвращаемого функцией think(), функции thought().

Объявление указателя на функцию. Если объявлены указатели на типы данных, то в этом объявлении должно быть четко отмечено, на какой тип указывает тот или иной указатель. Аналогично указатель на функцию должен указывать, на какой тип функции ссылается указатель. Это означает, что такое объявление должно идентифицировать тип данных, возвращаемых функцией, равно как и сигнатуру функции (список ее аргументов). Другими словами, такое объявление должно сообщать нам те же сведения о функции, которые уже содержатся в прототипе этой функции. Рассмотрим функцию со следующим прототипом:

double pam(int); // прототип

Вот как выглядит объявление соответствующего типа указателя:

*double (*pf)(int);*

// pf указывает на функцию, которая принимает один

// аргумент типа int и которая возвращает тип double

В объявлении необходимо заключить *pf в круглые скобки, чтобы соблюсти соответствующий приоритет операторов. Круглые скобки имеют более высокий приоритет, чем оператор *, следовательно, *pf(int) означает, что pf() — это функция, которая возвращает указатель, в то время как (*pf)(int) означает, что pf является указателем на функцию:

```
double (*pf) (int) ;
// pf указывает на функцию, которая возвращает значение
//типа double
double *pf(int);
// pf() является Функцией, которая возвращает указатель
// на значение типа double
```

После того как правильно объявлен указатель pf, можно присвоить ему адрес подходящей функции:

```
double pam(int); double (*pf) (int) ;
pf = pam;     // теперь pf указывает на функцию pam ()
```

Функция *pam()* должна соответствовать pf как по сигнатуре, так и по типу. Компилятор отвергает противоречивые операции присваивания, содержащие подобного рода несоответствия:

```
double ned(double);
int ted(int);
double (*pf) (int);
pf = ned;     // неправильно — несоответствие сигнатур
pf = ted;     // неправильно — несоответствие возвращаемых
//типов
```

Использование указателя для вызова функции. Поскольку pf уже объявлен, все что нужно сделать, — это воспользоваться (*pf), как если бы это было имя функции:

```
double pam(int);
double (*pf)(int);
pf = pam; // pf сейчас указывает на функцию pam()
double x = pam(4);
//обратиться к pam(), используя для этой цели имя функции
double y = (*pf) (5) ;
//обратиться к pam(), используя для этой цели указатель pf
```

По существу, C++ также позволяет вам использовать указатель pf, как если бы он был именем функции:

```
double y = pf(5);
// также обращается к // функции pam (), используя для
// этой цели указатель pf.
```

Итак, имя функции C++ ведет себя как адрес этой функции. Путем использования аргумента типа функции, являющегося указателем на другую функцию, можно передать функции имя второй функции, если требуется, чтобы к ней обратилась первая функция.

Шаблоны функций

Современные компиляторы языка C++ реализуют одно из его новейших средств - шаблоны функций. Шаблоны функций представляют собой обобщенное описание функций. Другими словами, они определяют функцию, используя термины обобщенных типов, вместо которых могут быть подставлены конкретные типы, такие как, например, int или double. Передавая тип в качестве параметра, можно заставить компиля-

тор генерировать функцию этого конкретного типа. Поскольку шаблоны позволяют программировать на основе обобщенных типов без использования специализации, этот процесс иногда называют обобщенным программированием. Поскольку типы представлены параметрами, шаблоны функций иногда называют параметризированными типами. Применение этого средства проиллюстрируем на простом примере обмена значений двух переменных. Предположим, мы уже определили функцию, которая производит обмен значений двух переменных типа **int.** Предположим, что вместо этого можно произвести обмен значений двух переменных типа **double.** Один из подходов к решению этой задачи состоит в сохранении исходного программного кода, однако при этом придется заменить каждый тип **int** на **double.** Если нужно произвести обмен значениями двух переменных типа **char,** можно снова воспользоваться этим методом. Тем не менее, чтобы выполнить такие незначительные изменения, придется затратить время, при этом отнюдь не исключена вероятность ошибки.

Средство шаблона функции в языке C++ автоматизирует этот процесс, обеспечивая высокую надежность и экономию времени.

Шаблоны функций предоставляют возможность давать определения функций на основе некоторого произвольного типа данных. Например, можно построить следующий шаблон обмена:

```
template <typename Any>
void Swap (Any &a, Any &b)
{
	Any temp;
	temp = a;
	a = b;
	b = temp;
}
```

Первая строка показывает, что устанавливается шаблон и произвольному типу данных присваивается имя **Any.** Обязательным условием является употребление ключевых слов **template и typename,** равно как и угловых скобок. Имя типа выбирается по собственному усмотрению, при этом необходимо соблюдать все правила использования имен, действующие в C++. Остальная часть программного кода описывает алгоритм обмена значениями типа **Any.** Шаблон не создает никаких функций. Вместо этого он предоставляет компилятору указания, касающиеся того, как определить функцию. Если необходимо, чтобы функция осуществляла обмен значениями типа int, то компилятор создаст функцию, соответствующую образцу шаблона, подставляя int вместо Any. Аналогично, если нужна функция, которая производит обмен значениями типа

double, компилятор будет руководствоваться требованиями шаблона, подставляя тип double вместо Any.

Использование шаблонов оправдано в тех случаях, когда определяются функции, применяющие один и тот же алгоритм к различным типам данных.

Чтобы сообщить компилятору о том, что нужна некоторая конкретная форма функции обмена, нужно воспользоваться в программе функцией с именем Swap(). Компилятор сначала проверит типы используемых аргументов, а затем создаст соответствующую функцию.

Перегрузка операций

Это один из методов, который позволяет придать операциям с объектами более привлекательный вид. *Перегрузка операций* представляет собой еще один пример полиморфизма C++. Как известно, C++ обеспечивает возможность определять несколько функций, имеющих одно и то же имя при условии, что у них разные сигнатуры (списки аргументов). Это и есть перегрузка функций, или функциональный полиморфизм. Он предназначен для того, чтобы предоставить возможность использовать одно и то же имя функции для выполнения одних и тех же базовых операций, даже если такая операция применяется по отношению к различным типам данных. Перегрузка операций (операторов) позволяет рассматривать операции (операторы) C++ сразу в нескольких смыслах. Фактически многие операции языка C++ (и языка C) уже перегружены изначально. Например, операция *, будучи примененной к адресу, дает значение, которое хранится по этому адресу. Однако, применяя операцию * к паре чисел, мы получаем произведение этих значений. C++ использует число и типы операндов, чтобы решить, какое действие предпринять.

C++ позволяет распространить перегрузку операций (операторов) на типы, определенные пользователем. Благодаря этому появляется возможность, например, использовать символ + для сложения двух объектов. Опять-таки, компилятор воспользуется числом и типом операндов, чтобы определить, каким определением операции сложения воспользоваться. Перегруженные операции часто делают вид программного кода более привычным. Например, сложение двух массивов — это обычная операция при выполнении вычислений. Обычно эта операция приобретает с помощью цикла for следующую компактную форму;

```
// поэлементное  сложение
for(int i  = 0; i < 20; i++)
     s[i] = s1[i] + s2[i];
```

Но в C++ вы можно определить тип, который представляет массивы и перегружает операцию + таким образом, что можно выполнить следующее действие:

```
// сложить два объекта типа массив
s = s1 + s2;
```

Такая простая форма записи операции сложения скрывает ее механику и подчеркивает самое существенное.

Чтобы перегрузить какую-либо операцию, лучше воспользоваться функцией специальной формы, получившей название *операторной.* Операторная функция имеет вид:

operator *op* (список_аргуменгов)

где *op* — это символ операции, подвергаемой перегрузке. Другими словами, operator+() перегружает операцию + (здесь *op* — это +), а operator*() перегружает операцию * (здесь *op* — это *). Операцией *op* может быть только полноценная операция C++; для этого недостаточно ввести новое обозначение и этим ограничиться. Например, нельзя пользоваться функцией operator@(), поскольку в C++ нет операции @. Но функция operator[]() перегрузит операцию [], так как [] — это операция, выполняющая индексацию массива.

Форматирование вывода с помощью cout

Операции вставки класса ostream преобразуют значения в текстовый вид. По умолчанию они форматируют значения следующим образом:

- Значение типа char, если оно представляет собой символ, который можно вывести, печатается как символ в поле, размер которого составляет один символ.
- Числовые значения целого типа выводятся как десятичные целые числа в поле, размер которого достаточен для вывода всех цифр и при необходимости для вывода знака "минус".
- Строки выводятся в поле, размер которого равен длине строки.
- Типы чисел с плавающей точкой выводятся в шести полях, замыкающие нули не выводятся. (Важно, что количество выводимых цифр никак не связано с точностью, с которой число хранится в памяти.) Число выводится в записи с фиксированной десятичной точкой или в научной записи - в зависимости от величины числа. В ча-

стности, научная запись используется в том случае, если показатель имеет значение 6 и больше или -5 и меньше. Снова поле является достаточно большим, чтобы поместить число и при необходимости знак "минус". Поведение, заданное по умолчанию, отвечает использованию функции fprintf() стандартной библиотеки C со спецификатором %g.

Поскольку каждое значение выводится в поле, размер которого соответствует этому значению, необходимо явно указывать пробелы между значениями, иначе последовательные значения сольются вместе.

Изменение системы счисления при выводе

Класс ostream наследуется из класса ios, который, в свою очередь, наследуется из класса ios_base. В классе ios_base сохраняется информация, необходимая для форматирования вывода. Например, определенные биты в одном из элементов класса показывают, какая система счисления используется для вывода, а другой элемент класса определяет ширину поля. Используя манипуляторы, можно управлять системой счисления при выводе целых чисел. С помощью функций-элементов класса ios_base можно контролировать ширину и количество полей, отводимых для отображения цифр, находящихся справа от десятичной точки. Поскольку класс ios_base является косвенным базовым классом для класса ostream, можно использовать его методы вместе с такими объектами класса ostream (или его потомков), как cout.

Посмотрим, например, как установить систему счисления для вывода целых чисел. Чтобы вывести целые числа в десятичной, шестнадцатиричной или восьмеричной системе счисления, можно воспользоваться манипуляторами dec, hex или oct. Например, при вызове функции *hex(cout)* устанавливается шестнадцатиричная система счисления для форматирования в объекте cout. После однократного вызова программа будет выводить числа в шестнадцатиричной форме до тех пор, пока состояние форматирования не будет изменено.

Несмотря на то что манипуляторы реально являются функциями, их обычно используют следующим образом: *cout « hex.*

Класс ostream перегружает операцию « так, чтобы ее использование было эквивалентно вызову функции hex(cout).

Установка ширины полей. Можно воспользоваться функцией-элементом width, чтобы поместить различные числа в поля одинаковой ширины. Метод имеет следующие прототипы:

```
int width();
```

```
int width(int i);
```

Первая форма возвращает текущую установку для ширины полей. Вторая устанавливает ширину поля равной *i* пробелам и возвращает предыдущее значение ширины поля. Это позволяет сохранить предыдущее значение в том случае, если оно может понадобиться в будущем.

Метод width() влияет только на следующий вывод, а после него ширина поля возвращается к предыдущему значению. Например, рассмотрим следующее выражение:

```
cout « '#' ;
cout.width(12);
cout « 12 « "#" « 24 « "#\n";
```

Выражение отображает на экране следующую информацию:

```
#                  12#24#
```

Число 12 выводится в поле из 12 символов, выровненное по правой стороне. Это называется выравниванием справа. После этого ширина поля возвращается к значению, заданному по умолчанию, и два символа # и число 24 выводятся в полях необходимого для них размера.

Язык С++ никогда не урезает отображаемые данные, поэтому, если попытаться вывести на печать значение из семи символов в поле из двух символов, С++ расширит поле, чтобы вместить данные.

Символы – заполнители. По умолчанию cout заполняет поля пробелами. Для изменения символа-заполнителя можно использовать метод fill(). Например, в результате вызова метода *cout.fill('*')* символ-заполнитель заменяется на звездочку. Это может быть удобно, например, для печати чеков, чтобы получатель не мог добавить несколько цифр к числу. в отличие от ширины полей, новый символ заполнения будет действовать до тех пор, пока его не изменить явно.

Установка точности при выводе чисел с плавающей точкой. Значение точности чисел с плавающей точкой зависит от режима вывода. В режиме, заданном по умолчанию, она обозначает количество выводимых цифр. В фиксированном или научном режиме, точность обозначает количество цифр выводимых после десятичной точки. Точность, установленная для С++ по умолчанию, равна 6 (замыкающие нули отбрасываются). Функция-элемент precision() позволяет выбрать другие значения. Например, выражение *cout.precision(2)* заставляет объект cout использовать точность, равную 2. В

отличие от случая с функцией width(), но подобно случаю с функцией fill(), новые установки точности остаются действовать до их явного обновления.

Вывод замыкающих нулей и десятичной точки. Некоторые формы вывода, например цены или цифры в колонках, выглядят лучше, если замыкающие нули присутствуют. Семейство классов iostream не включает функцию, которая могла бы выполнить эту задачу. Однако класс ios_base содержит функцию setf() (сокращение от слов set flag — установить флаг), которая управляет определенными свойствами форматирования. Класс также определяет несколько констант, которые можно использовать в качестве аргументов этой функции. Например, вызов функции *cout.setf(ios_base::showpoint)* заставляет cout вывести десятичную точку.

showpoint - это статическая константа в диапазоне доступа класса ios_base, определенная в объявлении класса. Диапазон доступа класса указывает, что нужно использовать оператор диапазона доступа (::) с именем константы, если это имя применяется за пределами метода класса. Таким образом, ios_base::showpoint — это имя константы, определенной в классе ios_base.

Стандартные манипуляторы. Использование setf() — это не самый удобный для пользователя подход к форматированию, поэтому C++ имеет несколько манипуляторов, которые вызывают setf(), автоматически задавая необходимые аргументы. Эти манипуляторы работают так же, как hex. Например, оператор *cout « left « fixed* задает выравнивание слева и запись с десятичной точкой. В табл. 16.3 перечислены различные манипуляторы.

Таблица 1. Некоторые стандартные манипуляторы

Манипулятор	Вызов
boolalpha	setf(ios_base::boolalpha)
noboolalpha	unsetf(ios_base::noboolalpha)
showbase	setf(ios_base::showbase)
noshowbase	unsetf(ios_base::showbase)
showpoint	setf(ios_base::showpoint)
noshowpoint	unsetf(ios_base::showpoint)
Showpos	setf(ios_base::showpos)
noshowpos	unsetf(ios_base::showpos)
uppercase	setf(ios_base::uppercase)
nouppercase	unsetf(ios_base::uppercase)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios^base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios base::basefield)
fixed	setf (ios_base:-.fixed, iosjbase: : f loatf ield)
scientific	setf(ios_base::scientific, ios_base::floatfield)

Литература

1. Бьорн Страуструп Б. Язык программирования С++. -Бином, 1999. -990с.
2. Прата С.. Язык программирования С++. – иев, DiaSoft, 2001 -636с.

Программирование

Пособие по учебной практике
для студентов 1-го курса
специальности «Прикладная математика»

Т.И. Петрушина, И.Н. Лисицына

Видано в авторській редакції

Підп. до друку 19.05.2010. Формат 60х84/8.
Гарн. Таймс. Тираж 50 прим.

Редакційно-видавничий Центр
Одеського національного університету
імені І.І. Мечникова,
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39