МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ
И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

УТВЕРЖДАЮ
Заведующий кафедрой АОИ
д-р техн. наук, проф.
_____Ю.П. Ехлаков
«___»_____2016 г.

Информатика и программирование

методические указания к лабораторным работам и практическим занятиям
для студентов направления
09.03.04 – «Программная инженерия»

Разработчик
ст. преподаватель каф. АОИ
_____Н.В. Пермякова
«___»_____2016 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Основная цель курса – развитие теоретических представлений и практических навыков работы с информацией, хранящейся или обрабатываемой в вычислительных системах, способам представления данных и их обработки с помощью современных информационных технологий, формирование у студентов объектно-ориентированного мышления, обучение объектно-ориентированному (ОО) подходу к анализу предметной области и использованию объектно-ориентированной методологии программирования при разработке программных продуктов.

Для достижения указанной цели при выполнении лабораторных работ решаются следующие задачи:

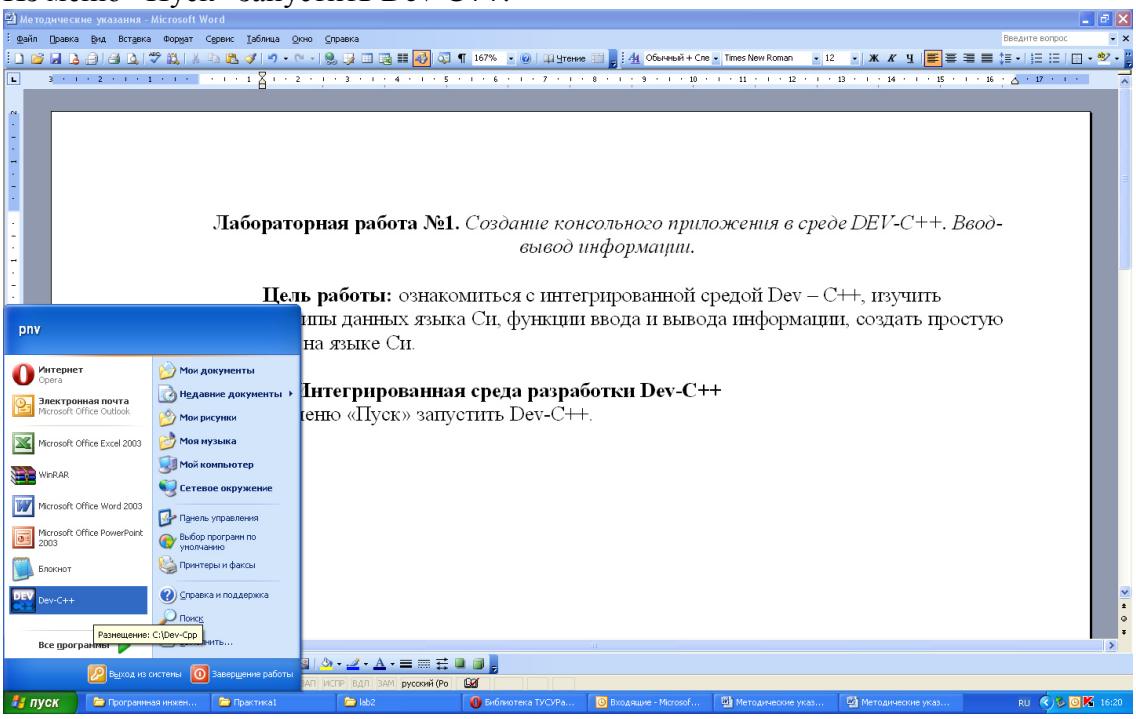
- формирование у студента знаний основных понятий, концепции, принципов и теорий, связанные с информатикой, понятия количества информации, типов систем счисления, структуры операционных систем, устройства файловых систем, основ архитектуры компьютера, способов представления алгоритмов, основных принципов структурного программирования;
- получение студентами навыков осуществления операций преобразования и математических операций над данными, представленными в разных системах счисления, представления алгоритмов, программирования на языке высокого уровня;
- обучение студентов владению языками структурного программирования, математическим аппаратом систем счисления, навыками использования прикладных программ, навыками разработки и отладки программ на алгоритмических языках программирования;
- изучение техники объектно-ориентированного анализа;
- изучение приемов объектно-ориентированного программирования.

Лабораторная работа №1. *Ввод-вывод данных. Проверка условий. Геометрия на плоскости.*

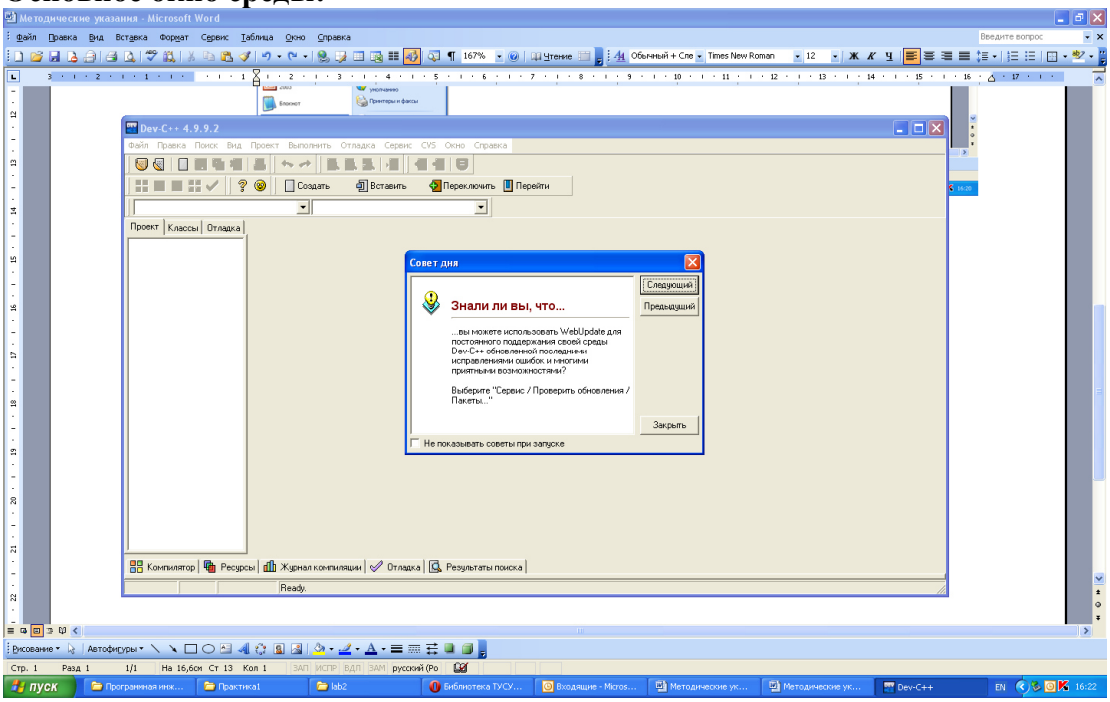
Цель работы: ознакомиться с интегрированной средой Dev – C++, изучить основные типы данных языка Си, функции ввода и вывода информации, создать простую программу на языке Си, ознакомиться с возможностями функции *scanf()*. Научиться составлять условные алгоритмы на примере алгоритмов проверки ошибок ввода данных и проверки вхождения точки в заданную область. Реализовать алгоритм на языке Си.

Интегрированная среда разработки Dev-C++

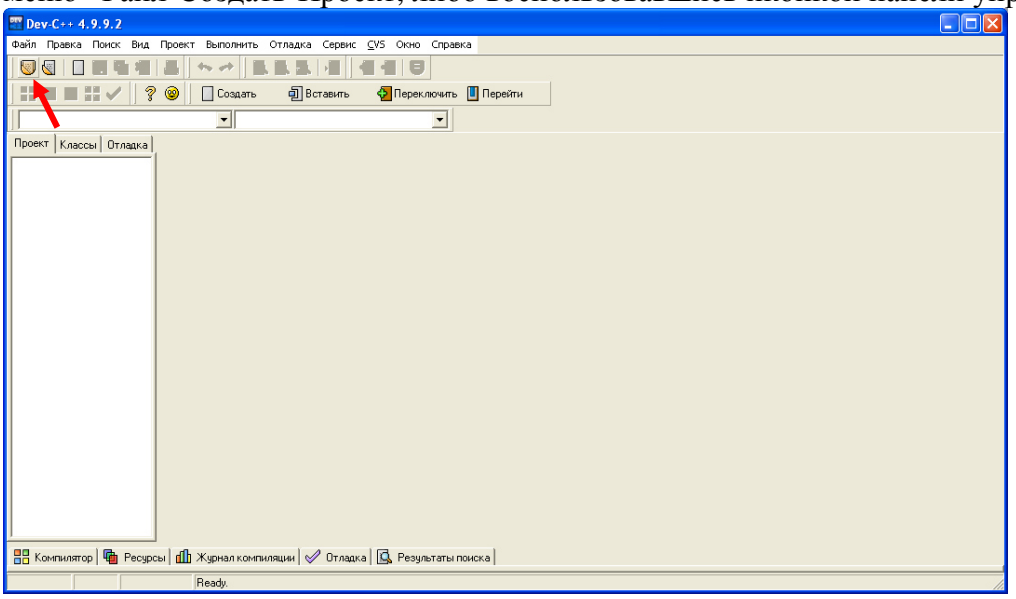
Из меню «Пуск» запустить Dev-C++.



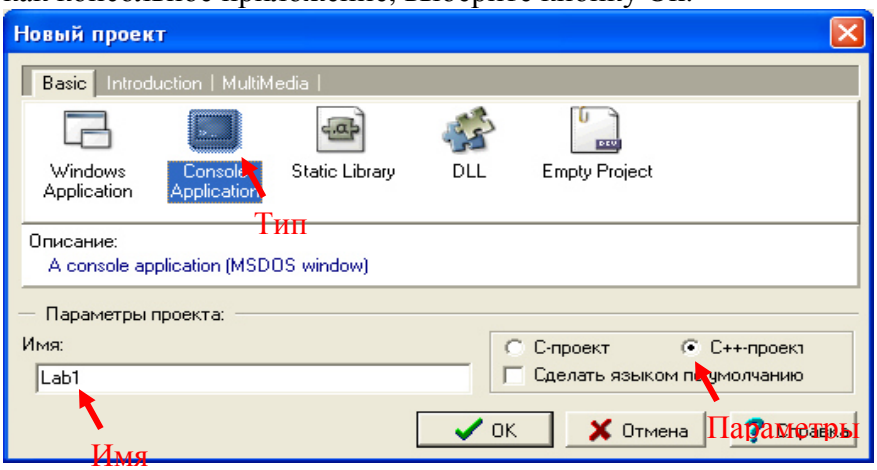
Основное окно среды:



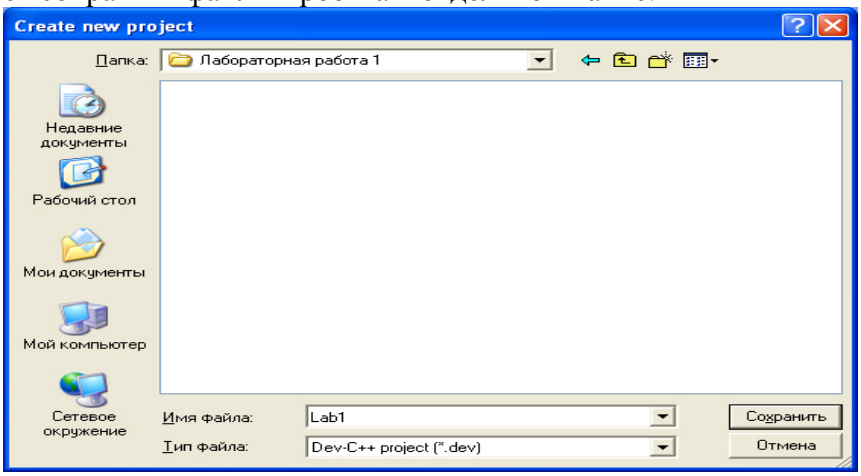
Выберите пункт меню «Создание проекта», либо используя навигацию по системе меню Файл-Создать-Проект, либо воспользовавшись иконкой панели управления:



Дайте проекту имя (желательно использовать символы английского алфавита), выберите параметры проекта (Си или Си++), рекомендуется Си++, определите тип проекта как консольное приложение, выберите кнопку Ok:

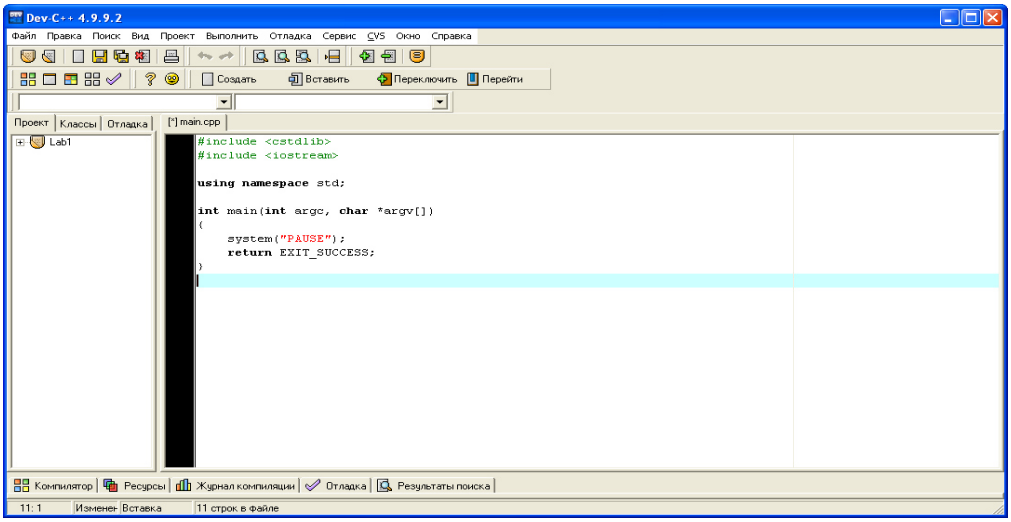


Далее среда предлагает выбрать место для сохранения файлов проекта. Рекомендуется сохранять файлы проекта в отдельной папке:



Для сохранения файлов доступны Рабочий стол и диск Temp.
После выбора места нажмите кнопку «Сохранить».

После выполнения всех выше описанных действий среда создаст шаблон простейшего консольного приложения:



Структура программы на языке Си

1. #include <cstdlib> - подключить заголовочный файл *cstdlib.h*.
2. #include <iostream> - подключить заголовочный файл *iostream.h*.
3. using name space std – использовать стандартное пространство имен
*4. int main(int argc, char *argv[])* – имя функции. Любая программа на языке Си состоит из одной или нескольких функций. В написанном шаблоне функция одна – *main( )*. Функция с именем ***main*** обязательно должна быть в ***любой исполняемой программе***.
5. { - начало тела функции
6. system("pause") – вызов функции *system* с аргументом *"pause"*. Функция реализует ожидание нажатия клавиши.
7. оператор *return* с аргументом *EXIT_SUCCESS* – завершение функции main с кодом 0.
8. } – конец функции ***main.***

Простые типы данных Си

Для представления целых величин в Си предусмотрены следующие типы данных:

Тип char. Занимает в памяти 1 байт. Используется для представления символов и целых чисел от 0 до 255 (-128 до 127).

Тип int. Занимает в памяти 4 байта. Используется для представления целых чисел в диапазоне -2 147 483 648 до 2 147 483 647.

Тип float. Занимает в памяти 4 байта. Используется для представления чисел с плавающей точкой. от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{38}$. Точность вычислений до 7 знаков после запятой.

Тип double. Занимает в памяти 8 байт. Используется для представления чисел с плавающей точкой. от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{308}$. Точность вычислений до 15 знаков после запятой.

Тип void – пустой тип. Используется для описания функций.

Тип bool – логический тип. Может принимать 2 значения true или false.

Основные операторы Си

Оператор - это лексема, которая переключает некоторые вычисления, когда применяется к переменной или к другому объекту в выражении. Язык Си представляет большой набор операторов арифметических и логических операторов.

Таблица 4.1. Унарные операторы языка Си

Код оператора	Название	Результат операции
&	адресный оператор	выражение *&x* - адрес переменной *x*
+	унарный плюс	+5 – положительная константа
-	унарный минус	-4 – отрицательная константа, *-x* – значение переменной *x* с обратным знаком
!	логическое отрицание	*!x* принимает значение 0 (лжи), если x имеет ненулевое (истинное) значение и наоборот
++	префиксное/ постфиксное увеличение	*int x = 5; ++x;* увеличит *x* на единицу; *int x = 5; x++;* увеличит *x* на единицу
--	префиксное/ постфиксное уменьшение	*int x = 5; --x;* уменьшит x на единицу; int x = 5; x--; увеличит x на единицу

Таблица 4.2. Бинарные операторы языка Си

Код оператора	Название	Результат операции
Аддитивные операторы		

+	бинарный плюс	вычисление суммы, например:
		int x = 2,y = 1,z;
		z = x+y;

-	бинарный минус	вычисление разности, например:
		int x = 2,y = 1,z;
		z = x-y;

Мультипликативные операторы		

*	умножение	вычисление произведения, например:
		int x = 2,y = 1,z;
		z = x*y;

/	деление	вычисление частного, например:
		int x = 12,y = 2,z;
		z = x/y;

%	остаток	вычисление остатка от деления, например:
		int x = 12,y = 7,z;
		z = x%y;

Логические операторы		
&&	логическое AND (И)	проверка условий, связанных логическим И
||	логическое OR (ИЛИ)	проверка условий, связанных логическим ИЛИ
Операторы присваивания		
=	присваивание	присвоить переменной заданное значение или значение другой переменной
Операторы отношения		
<	меньше чем	x<y, x меньше y
>	больше чем	x>y, x больше y
<=	меньше чем или равно	x<=y, x меньше или равно y
>=	больше чем или равно	x>=y, x больше или равно y
Операторы эквивалентности		
= =	равно	x= =y, x равно y
!=	не равно	x!=y, x не равно y
,	оператор перечисления	выполнить разделенные оператором действия слева направо, например y+=5,x-=4,y+=x;

Библиотека математических функций math.h

Си поддерживает множество математических функций, прототипы которых описаны в файле *math.h.* Познакомимся с некоторыми из них.

abs(int x) возвращает модуль целого числа *x*.
acos(double x) возвращает арккосинус числа *x* в радианах.
asin(double x) возвращает арксинус числа *x* в радианах.
atan(double x) возвращает арктангенс числа *x* в радианах.
*atof(char *s, double x)* преобразует строку *s* в вещественное число *x.*
cos(double x) возвращает косинус числа *x* (*x* задано в радианах)
ceil(double x) округляет число *x* в большую сторону
exp(double x) возвращает экспоненту числа *x.*
fabs(double x) возвращает модуль вещественного числа *x.*
sin(double x) возвращает синус числа *x* (*x* задано в радианах).
sqrt(double x) возвращает квадрат числа *x.*
tan(double x) возвращает тангенс числа *x* (*x* задано в радианах).
floor(double x) округляет число *x* в меньшую сторону
fmod(double x, double y) возвращает остаток от деления числа *x* на число *y.*
hipot(double x, double y) возвращает квадрат суммы числа *x* и числа *y.*
log(double x) возвращает натуральный логарифм числа *x.*
log10(double x) возвращает десятичный логарифм числа *x.*
modf(double x,double& y) возвращает дробную часть числа *x,* по адресу *y* записывается целая часть исходного числа *x.*
pow(double x, double y) возвращает *x* в степени *y.*

Для использования всех вышеперечисленных функций подключите библиотеку *math.h:*

#include< cmath>

Основные правила программирования на Си

- Исполняемая функция обязательно должна носить имя *main*
- Все используемые в программе переменные должны быть описаны перед использованием
- Количество открывающихся фигурных скобок должно быть равно количеству закрывающихся

Вывод информации в Си

Для вывода информации на экран Си предоставляет множество возможностей. Есть функции, выводящие на экран только строки, только целые или вещественные числа. Функция *printf* может использоваться для вывода на экран информации любого типа.

Описание функции:

printf(Управляющая строка, <аргумент1, аргумент2, …>);

Управляющая строка записывается в двойных кавычках и содержит информацию двух типов:

– печатаемые символы (константная строка);

– идентификаторы данных (спецификаторы формата)

Функция принимает список аргументов и применяет к каждому спецификатор формата. Количество спецификаторов формата и аргументов должно быть одинаковым.

Основные спецификаторы формата –

%d – целое десятичное число;

%c – один символ;

%s – строка символов;

%e – экспоненциальная запись числа с плавающей точкой;

%f – десятичная запись числа с плавающей точкой;

%u - десятичное число без знака;

%o - целое восьмеричное число без знака;

%x - целое шестнадцатеричное число без знака.

Помимо этого в спецификаторах используются модификаторы, форматирующие выводимую информацию. Рассмотрим применение модификаторов на спецификаторе *%f*. Аналогично форматируется информация других типов.

Модификатор состоит из двух чисел, разделенных точкой и может иметь лидирующий знак *«-»*. Записывается модификатор после знака *«%»*, обозначающего начало спецификатора. В общем виде модификатор выглядит следующим образом: *%m.n*спецификатор. Первое число *m* задает ширину поля вывода для всего значения. Второе число *n* используется для форматируемого вывода чисел с плавающей точкой и задает количество дробной части числа, выводимых на экран. Отсутствие знака *«-»* говорит о том, что вывод будет отформатирован по правой границе поля вывода, присутствие – форматирование по левой границе поля вывода.

При записи спецификатора в следующем виде - *%10.4f* – все выводимое вещественное число запишется в поле из десяти символов. Дробная часть числа будет состоять из 4 знаков.

Например:

```
…
int old = 23;
float key = 15.164;
char String[15] = “Простая программа”;
printf(“/%10d/”,old); // выведется /       23/
printf(“/%-10d/”,old); // выведется /23       /
printf(“/%10.1f/”,key); // выведется /      15.2/
printf(“/%-10.4f/”,key); //выведется /15.1640  /
printf(“/%5.5s/”,string); // выведется /Прост/
printf(“/%-30s/”,string); // выведется / Простая программа          /
…
```

Для перевода вывода на другую строку используется специальный символ *‘/n’*.

```
…
#define PI 3.141
printf (“Пример использования функции printf: \n число PI = %8.2f”, PI);
…
```

На экране:

```
Пример использования функции printf:
число PI =  3.14
```

При наличии в строке вывода специальных символов, используемых для форматирования (например, %,\ и тому подобное) эти символы дублируются. Первый символ интерпре-

тируется как специальный, а второй такой символ, говорит о том, что это часть выводимой информации.

Ввод информации

Для ввода информации Си предлагает наиболее общую функцию (функцию работающую с разнотипными данными) *scanf*

Описание функции:

scanf(спецификатор формата, указатель на переменную);

В функции используются те же спецификаторы формата, что и в функции *printf*.

Обратите внимание. Имя массива является указателем, поэтому при вводе строк перед именем строки не пишется знак *&*. При вводе строки с помощью функции *scanf* строка вводится до первого встреченного пробела. Вся остальная часть строки обрезается.

Например:

```
…
char name[20];
scanf(“%s”,name);  // ввод строкового массива.
int n;
scanf(“%d”,&n); // ввод целочисленной переменной n.
scanf(“%c”,&name[3]);  //ввод четвертого символа массива name.
```

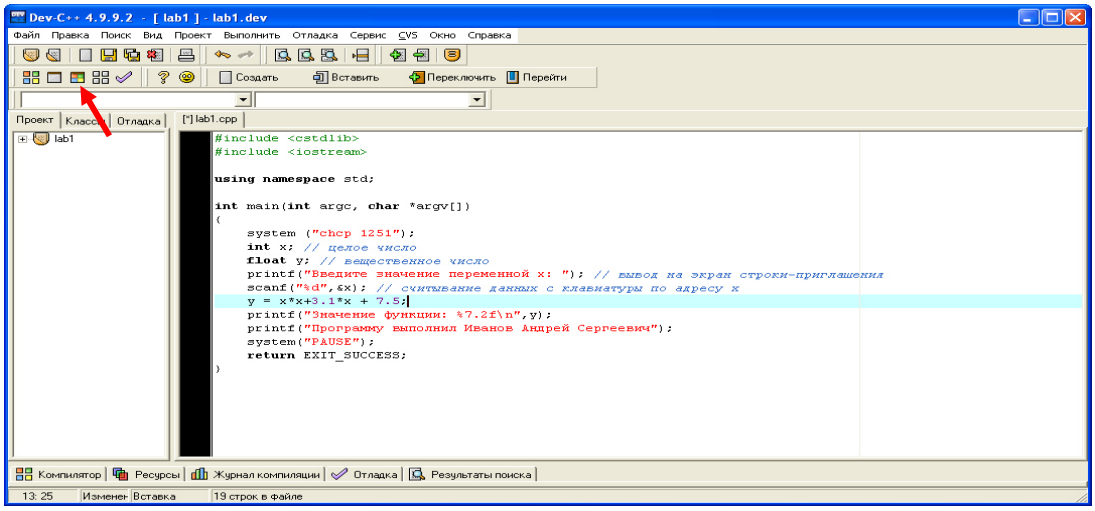
При одном вызове функции возможно ввести более одной переменной. В этом случае спецификаторы формата пишутся один за другим, без пробелов. Каждому спецификатору должен соответствовать свой адрес переменной. Например:

```
…
float x,y,z;
printf(“Введите значения переменных x,y и z: ”);
scanf(“%f%f%f”,&x,&y,&z);
…
```

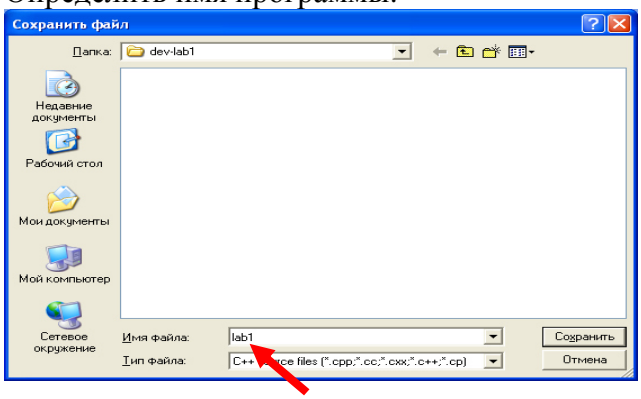
Пример программы

Задание: Ввести с клавиатуры целое число *x*. Вывести на экран значение функции $x^2 + 3.1x + 7.5$ и сообщение вида: «Программу выполнил ФИО»

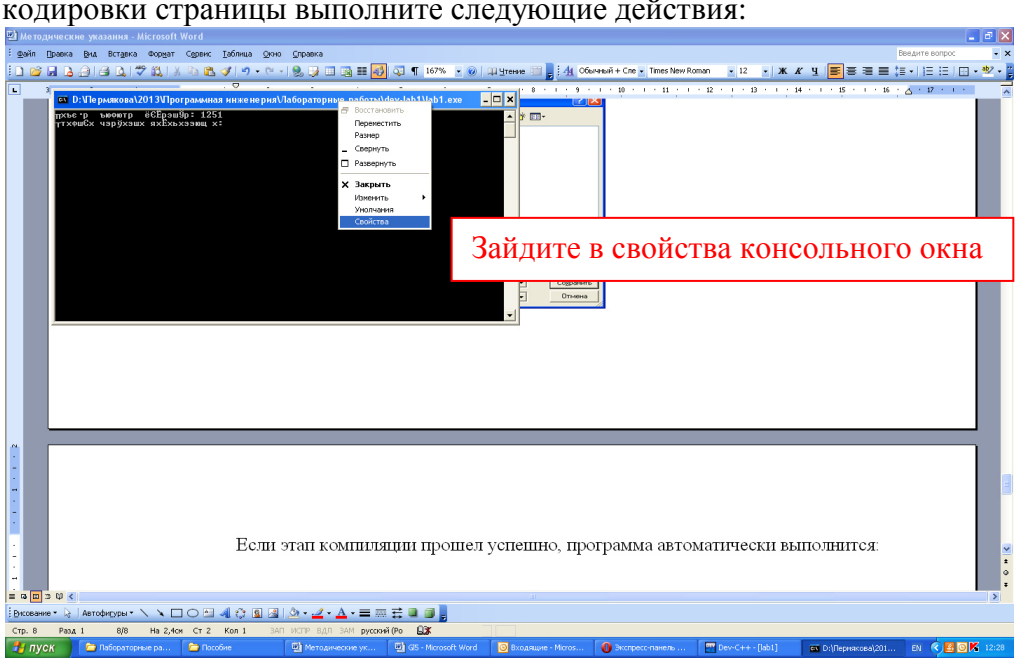
2.	Создание проекта	Запустить Dev – C++, создать новый проект, дополним код программы вызовом функции system ("chcp 1251"); - смена кодировки страницы
3.	Описание переменных	int x; float y;
4.	Ввод данных с клавиатуры	printf(“Введите значение переменной x: ”); scanf(“%d”,&x);
5.	Вычисление значения функции	y = x*x+3.1*x + 7.5;
6.	Вывод результата	printf(“Значение функции: %7.2f\n”,y);
7.	Вывод личных данных	printf(“Программу выполнил Иванов Андрей Сергеевич\n”);

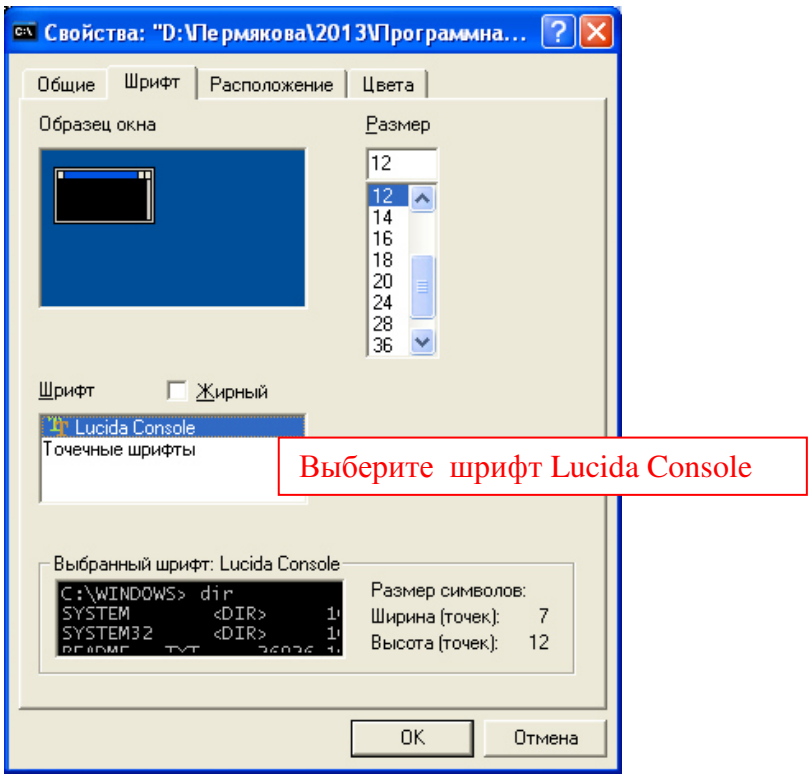


Определить имя программы:

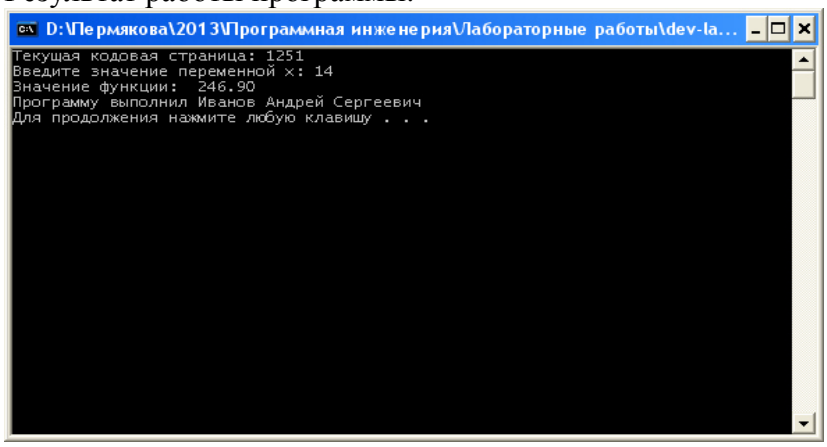


Если этап компиляции прошел успешно, программа автоматически выполнится. Для смены кодировки страницы выполните следующие действия:





Результат работы программы:



Разветвляющиеся (условные) алгоритмы

Не всем алгоритмам достаточно для выполнения конструкции следования. Рассмотрим следующий пример: $f(x)=\begin{cases}\frac{1}{x}, \forall x \neq 0\\ 1, x=0\end{cases}$. Алгоритм решения этой задачи может быть следующим:

Шаг 1. Задать x.

Шаг 2. ЕСЛИ x=0 ТО f := 1 ИНАЧЕ $f := \frac{1}{x}$.

КОНЕЦ

Эта управляющая конструкция носит название ***развилка (проверка условия, условная конструкция) –***

ЕСЛИ *условие* **ТО** *действия при истинном условии* **ИНАЧЕ** *действия при ложном условии*.

Условная конструкция в языке Си

Для организации ветвления алгоритма в Си используется оператор проверки условия ***if*** ***(***логическое выражение***)*** ***{***действия при истинном значении выражения***}*** ***else {***действия при ложном значении выражения***}***. Оператор *else* может отсутствовать, если это обусловлено алгоритмом.

Логическое условие может быть сложным, т.е. может состоять из нескольких условий, связанных между собой логическими операциями:

- «И» (конъюнкция), в Си оператор ***&&***, все логическое выражение считается истинным только в том случае, если истинны все простые выражения.
- «ИЛИ» (дизъюнкция), в Си оператор ||, все логическое выражение считается ложным только в том случае, если ложны все простые выражения.

Например, запишем следующее условие «Если переменная *x* меньше переменной *y* и переменная *x* меньше переменной *z*»: *x<y && x<z.* Следующее условие демонстрирует операцию дизъюнкции «Если переменная *m < 10* или переменная *m* равна переменной *x*» : *m<10 || m == x.*

Если в блоках программы, выполняющихся при истинности или ложности условия необходимо выполнить два и более действий, эти блоки определяются фигурными скобками. Приведем примеры.

– Если *x<y*, то вывести на экран значение суммы *x* и *y*, значение переменной *x* заменить на *10.*

```
if (x<y)
  { printf("Сумма ==> %d",x+y);
    x = 10;
  }
```

– Если *x* не равно *y* вывести на экран соответствующее сообщение, в противном случае вывести на экран значения переменных переменную *x* увеличить в два раза.

```
if(x!=y)
  printf("Переменные имеют неравные значения");
else  {
        printf("x = %d, y = %d\n",x,y);
       x*=2;
      }
```

Возможности функции *scanf()*

При некорректном вводе (введены данные, несовпадающие с указанным спецификатором) не возникает ошибка выполнения. Но при этом дальнейшая работа программы не предсказуема. Такие ошибки можно отследить, проанализировав результат работы функции *scanf*. При успешном вводе результат работы функции – количество введенных верно полей. Например, проверку ошибок ввода можно выполнить, следующим образом:

```
…
int y,n;
printf("Введите значение переменной n: ");
y = scanf("%d",&n);
if (y!=1) {printf("Введены неверные данные…\n ");
            system("pause");
```

```
            exit(0);
            }
else { …}
```

В рассмотренном примере количество вводимых полей – 1 (вводится одна переменная *n*). Следовательно, если переменная *y* не принимает значение один – произошла ошибка ввода. На экран выводятся соответствующие сообщения. Программа ожидает нажатия клавиши и заканчивает работу (*exit*). В случае успешной работы выполняется часть программы, описанная в блоке *else*.

При одном вызове функции возможно ввести более одной переменной. В этом случае спецификаторы формата пишутся один за другим, без пробелов. Каждому спецификатору должен соответствовать свой адрес переменной. При этом *scanf* при успешном вводе возвращает количество успешно считанных полей. Например:

```
…
float x,y,z;
printf("Введите значения переменных x,y и z: ");
int m = scanf("%f%f%f",&x,&y,&z);
…
```

После успешного выполнения программы значение переменной *m* примет значение 3.

Задачи проверки вхождения точки с заданными координатами в ограниченную область.

Проверка расположения точки с координатами (x,y) относительно прямой (рис.2.1).

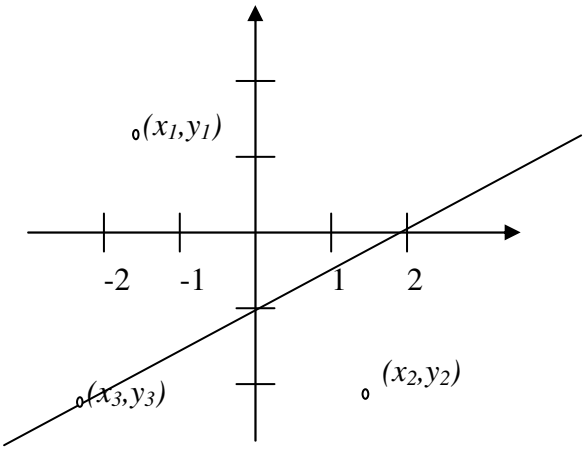


Рис. 1.1. Проверка расположения

Пусть уравнение прямой задано в каноническом виде $y = ax+b$. Тогда, все точки, лежащие **на линии** прямой подчиняются условию $y = ax + b$. Все точки, лежащие **ниже линии** прямой подчиняются условию $y < ax + b$, это условие выполняется для точки с координатами (x_1, y_1). Все точки, лежащие **выше линии** прямой подчиняются условию $y > ax + b$. Тогда:

- $y_3 = ax_3 + b$.
- $y_2 < ax_2 + b$.
- $y_1 > ax_1 + b$.

Для представленного рисунка составим уравнение прямой по двум заданным точкам: прямая проходит через точки с координатами (0,-1) и (2,0). Найдем коэффициенты уравнения *a* и *b*.

Решим систему уравнений:

$$\begin{cases} -1 = a \cdot 0 + b \\ 0 = a \cdot 2 + b \end{cases}$$

$$\begin{cases} b = -1 \\ a = \dfrac{-b}{2} \end{cases}$$

$$\begin{cases} b = -1 \\ a = 0.5 \end{cases}$$

$$y = 0.5x - 1$$

Таким образом, проверить, местоположение точки с координатами *(x,y)* можно следующим образом:

```
…
if (y<0.5*x-1) printf(“Точка расположена ниже прямой”);
    else   if(y>0.5*x-1) ) printf(“Точка расположена выше прямой”);
           else ) printf(“Точка расположена на прямой”);
```

*Проверка расположения точки относительно окружности известного радиуса и с заданным центром(*рис. *).*

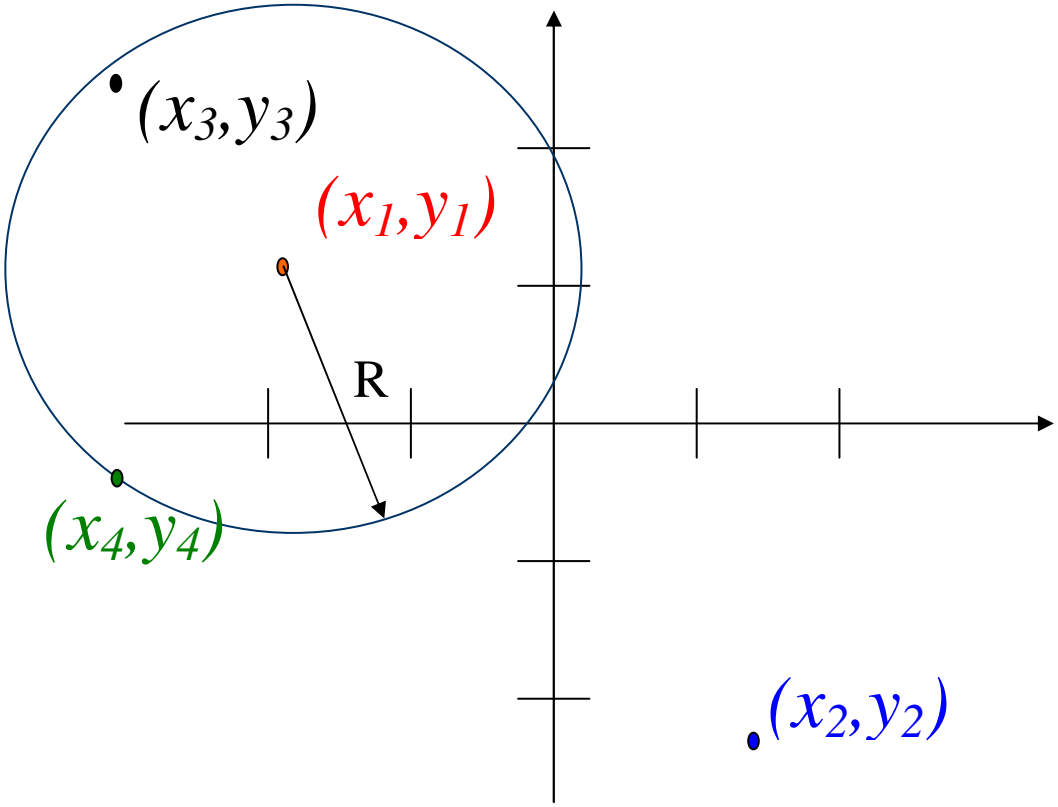


Рис. 1.2. Проверка расположения точки относительно окружности

Пусть общее уравнение окружности задано в виде:

$$R^2 = (x-x_1)^2+(y-y_1)^2$$

Тогда для точки с координатами (x_4,y_4) выполняется равенство:

$$R^2 = (x_4-x_1)^2+(y_4-y_1)^2$$

Это уравнение описывает все точки, лежащие на окружности. Для точки с координатами (x_2,y_2), и для всех точек, лежащих за окружностью, выполняется неравенство:

$$R^2 < (x_2-x_1)^2+(y_2-y_1)^2$$

То есть, радиус данной окружности меньше радиуса окружности с центром в точке (x_1,y_1), *на* которой лежит точка с координатами (x_2,y_2). Соответственно, для точки с координатами (x_3,y_3) выполняется неравенство -

$$R^2 > (x_3-x_1)^2+(y_3-y_1)^2$$

То есть, радиус данной окружности больше радиуса окружности с центром в точке (x_1,y_1), *на* которой лежит точка с координатами (x_3,y_3).

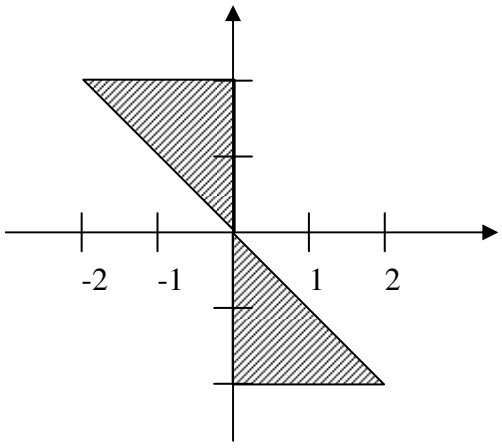


Рис. 1.3. Пример 1.

Построим условия вхождения точки в заданную область:

Уравнение прямой, на которой лежат гипотенузы прямоугольных треугольников, образующих фигуру ***y = -x.***

Разобьем фигуру на две части. Точка будет считаться принадлежащей фигуре, если она попадет в первую или вторую часть.

Первую (верхнюю часть) можно ограничить следующими условиями:

(y>=-x) и *(x<=0)* и *(y<=2)*

Первое условие описывает гипотенузу, второе и третье условие описывают катеты. Условия связаны между собой связками И (логическое умножение)

Вторую (нижнюю часть) можно ограничить условиями:

(y<=-x) и *(x>=0)* и *(y>=-2)*

Общее условие для двух частей будет выглядеть следующим образом:

Если *(y>=-x)* **и** *(x<=0)* **и** *(y<=2)* ***или*** *(y<=-x)* **и** *(x>=0)* **и** *(y>=-2)* **то** «Точка принадлежит заданной области», **иначе** «Точка не принадлежит заданной области»

Рассмотрим еще один пример:

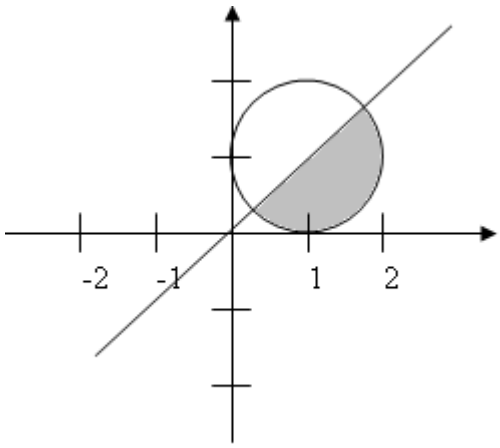


Рис. 1.4. Пример 2.

В этом случае уравнение прямой $y = x$. Уравнение окружности $1 = (x-1)^2+(y-1)^2$. Ограниченная область находится ниже прямой $(y<x)$ и внутри окружности - $1 > (x-1)^2+(y-1)^2$. Тогда общее условие будет выглядеть следующим образом:

> **Если** $y<x$ *и* $1 > (x-1)^2+(y-1)^2$ **то** «Точка принадлежит заданной области», **иначе** «Точка не принадлежит заданной области»

Порядок выполнения работы

1. Получить индивидуальное задание
2. По индивидуальному заданию определить типы и значения данных, являющиеся некорректными для задачи.
3. Составить и записать алгоритм решения задачи
4. Составить программу, реализующую алгоритм:
   - 4.1.Описать входные и выходные данные
   - 4.2.Ввести данные с клавиатуры
   - 4.3.Проверить входные данные
   - 4.4.Проверить условие вхождения точки в заданную область
   - 4.5.Вывести результат проверки на экран
   - 4.6.Вывести личные данные
   - 4.7.Выполнить компиляцию проекта
5. Написать отчет о проделанной работе.
6. Защитить работу

Задание оценивается в 7 баллов:
2 балла алгоритм,
1 балл код программы.
2 балла отчет.
2 балла защита.

Содержание отчета:

1.Текст задания
2. Вывод условий
3. Алгоритм
4. Тестирование программы

Контрольные вопросы

1. Какое имя носит исполняемая функция Си?

2. Дайте определение понятия «переменная»
3. Дайте определение понятия «идентификатор»
4. Сколько переменных требуется описать в программе, если необходимо решить следующую задачу – «С клавиатуры вводятся три числа, необходимо вывести на экран значение минимального из этих трех чисел»?
5. Какая функция используется в Си для ввода информации?
6. Какая функция используется в Си для вывода информации?
7. Какой тип данных Си соответствует спецификатору «%d»?
8. Какой тип данных Си соответствует спецификатору «%f»?
9. Переменная j описана в программе следующим образом:
   int j;
   Запишите функцию scanf для считывания значения в переменную j.
10. Переменная k описана в программе следующим образом:
    float k;
    Запишите функцию printf для вывода значения переменной k.
11. Что возвращает функция *scanf?*
12. Запишите функцию *scanf* для ввода трех переменных.
13. Что Вы понимаете под некорректными данными?
14. Какие данные будут некорректными для решения следующей задачи – “Даны длины трех сторон треугольника. Найдите площадь треугольника.”
15. Как в Си реализована условная конструкция структурного программирования?
16. Опишите синтаксис конструкции if else языка Си.
17. Какое значение примет переменная *m* после выполнения следующего фрагмента программы:

```
…
float i;
int j;
int m = scanf(“%f%d”,&i, &j);
…
```

если с клавиатуры были введены значения 3 2?

18. Какое значение примет переменная *m* после выполнения следующего фрагмента программы:

```
…
float i;
int j;
int m = scanf(“%f%d”,&i, &j);
…
```

если с клавиатуры были введены значения 3 d?

19. Какое значение примет переменная *x* после выполнения следующего фрагмента программы:

```
…
int x = 10;
int k = 12,z = 74;
if (k<z) x = 1; else x = 0;
…
```

20. Какое значение примет переменная *x* после выполнения следующего фрагмента программы:

```
…
int x = 10;
```

int k = 31,z = 22;
if (k<z) x = 1;
…

Лабораторная работа №2 *Вычисление суммы бесконечного ряда.*

Цель работы.

Программирование итерационных процессов. Использование конструкций циклов языка Си.

Конструкции циклов в языке Си

Цикл с фиксированным числом операций for

Цикл, это конструкция структурного программирования, повторяющая определенные действия (итерации) несколько раз. При заданном количестве итераций в Си используется конструкция *for*. Синтаксис:

for(секция инициализации значения; секция проверки условия; секция коррекции)

Значение, инициализируемое в первой секции, называется счетчиком цикла. Повторяемые действия называются телом цикла. Если тело цикла состоит из двух и действий, тело цикла заключается в фигурные скобки.

Секция инициализации выполняется один раз, поэтому может содержать описание переменной. На каждом шаге значение счетчика подставляется в условное выражение второй секции, если выражение истинно, то управление передается в тело цикла, в противном случае, управление передается за цикл. После каждого выполнения тела цикла выполняется операция из секции коррекции.

Например, цикл

for(int i=0;i<3;i++)
printf("%d \n",i);

будет работать следующим образом – счетчик *i* примет значение *0.* Условное выражение второй секции при таком значении счетчика истинно, на экран выведется значение переменной *i,* равное *0.* Управление передается в секцию коррекции и переменная *i* увеличивается на единицу. При этом условие все еще истинно, на экран выводится значение *1*. В ходе следующей итерации переменная-счетчик принимает значение *2*. Условное выражение остается истинным. На экран выводится значение счетчика. После этой итерации переменная *i* становится равной *3*. Условие при таком значении становится ложным, цикл заканчивает свою работу, управление передается следующей части программы.

Циклы while и do while

Существует множество задач, при выполнении которых циклические действия необходимо проводить до тех пор, пока истинно какое-либо условие. Для реализации таких алгоритмов возможно использовать циклы по условию *while* и *do while.*

Синтаксис: *while (условное выражение)*
{ тело цикла
}

Тело цикла *while* выполняется до тех пор, пока истинно условное выражение. Этот цикл называется циклом с предусловием. Цикл *while* может ни разу не выполниться (то есть, на входе в цикл условие ложно).

Синтаксис *do {*
тело цикла
} while (условное выражение)

Тело цикла *do while* выполняется пока условие истинно. Этот цикл называют циклом с постусловием. Такой цикл выполниться хотя бы один раз.

Операторы безусловной передачи управления continue и break

Оператор *break* досрочно завершает выполнение цикла. Управление передается оператору, следующему за циклом.

```
int n =15;
for(int i=0;i<n;i++)
{ int z = rand()%200;
if (z>100) break;}
```

Цикл должен выполняться *15* раз. В переменную *z* записывается случайное значение в интервале от *0* до *199* (функция *rand(), случайные числа от 0 до RAND_MAX (32767)*). Если получено случайное значение, большее *100*, цикл заканчивает свою работу.

Оператор *continue* пропускает все последующие операторы тела цикла и передает управление на начало цикла.

```
int f = 1;
do
{
int z = rand()%100;
if (z>30) continue;
if (z<10) f = 0;
printf("%d",z);
} while(f);
```

Описанный выше цикл работает следующим образом – цикл будет работать, пока переменная f равна единице. Если полученное случайное значение будет больше *30*, то вторая проверка условия и функция печати полученного значения будут пропущены. Если полученное значение будет меньше *10*, то переменной f будет присвоено значение *0*. Значение выведется на экран и цикл закончит свою работу.

В итоге, на экран будут выводиться числа, не больше тридцати и как только будет получено число, меньшее десяти, цикл закончит свою работу.

Вычисление суммы бесконечного ряда

Задание. Найти сумму ряда с заданной точностью. Точность и значение переменной *x* вводить с клавиатуры. Осуществить проверку ошибок ввода. Считать ошибочными значения *x*, которые приводят к расхождению ряда.

Найти сумму бесконечного ряда $\sum_{n=1}^{\infty} \frac{(-1)^n x^{2n+1}}{(n+2)!}$ с заданной точностью *e*.

Решение. Определим, что значит, найти сумму с заданной точностью. По виду общего элемента ряда $\frac{x^{2n+1}}{(n+2)!}$ (обозначим q_n) видно, что элементы ряда при определенных значениях *x* убывают при увеличении *n*. Таким образом, начиная с какого-то *n*, элементы ряда будут меньше заданной точности. А это, в свою очередь, говорит о том, что, начиная с этого элемента, приращение суммы никогда не станет больше заданной точности.

Таким образом, найти сумму бесконечного ряда с заданной точностью $0<\varepsilon<1$, значит просуммировать все значения , $q_1, q_2, \ldots q_n$, пока не найдется такое *n*, при котором q_n станет меньше заданного ε. Для того, чтобы составленный алгоритм был наиболее эффективным принято выражать q_{i+1} член ряда через q_i. Для вывода итерационной формулы разделим q_i на q_{i+1}:

$$\frac{q_i}{q_{i+1}} = \frac{x^{2i+1}}{(i+2)!} : \frac{x^{2i+3}}{(i+3)!} = \frac{(i+3)!}{(i+2)!\,x^2} = \frac{i+3}{x^2}.$$

Из полученного отношения выразим q_{i+1}

$$q_{i+1} = q_i \cdot \frac{x^2}{i+3}$$

При программировании $(-1)^n$ воспользуемся свойством сложения: если элемент ряда положительный, то прибавим его к общей сумме, иначе отнимем его от общей суммы.

Тогда алгоритм вычисления суммы бесконечного ряда можно записать следующим образом:

1. Задать точность ε, задать $S = 0$;
2. $n = 1$;
3. Задать значение *x*
4. Задать значение *overflow = false*
5. Вычислить значение q при $n = 1$, $q = \frac{x^3}{6}$
6. Вычислить значение $q1 = \frac{q \cdot x^2}{n+3}$.
7. **ПОКА** ($q >= \varepsilon$)
   - 7.1.**ЕСЛИ** n-четное **ТО** $S = S + q$
     ИНАЧЕ $S = S - q$
     КОНЕЦ ЕСЛИ
   - 7.2. $q = q1$;
   - 7.3. **ЕСЛИ** в переменной q возникает переполнение **ТО** *overflow = true,* перейти на 8.
   - 7.3. $n = n + 1$
   - 7.4. $q1 = \frac{q \cdot x^2}{n+3}$

   КОНЕЦ ЦИКЛА
8. **ЕСЛИ** *overflow* **ТО** Печать «Ряд расходящийся»
   ИНАЧЕ Печать количества итераций $n-1$, значение суммы S.
9. **Конец**

Порядок выполнения работы

1. Получить индивидуальный вариант задания.
2. Если необходимо, преобразовать исходную формулу ряда.
3. Написать и протестировать программу вычисления суммы бесконечного ряда.
4. Написать отчет.
5. Защитить работу.

Содержание отчета

Индивидуальное задание
Математические преобразования исходной формулы
Описание алгоритма
Результаты тестирования программы.

Задание оценивается в **7** баллов:

2 балла – алгоритм
1 балл – код программы
2 балла – отчет
2 балла – защита.

Лабораторная работа №3. *Статические массивы в Си. Алгоритмы поиска*

Цель работы: изучение способов работы со статическими одномерными массивами.

Статические массивы в Си.

Си поддерживает как одномерные, так и многомерные массивы. Под массивом будем понимать переменную, которая имея одно имя, может сохранять множество значений. Поскольку это переменная, то перед использованием ее в программе ее необходимо объявить. Общий синтаксис:

Тип_данных Имя_массива[<количество элементов в массиве>];

При объявлении массива может происходить и инициализация

Тип_данных Имя_массива[<количество элементов в массиве>]={<набор значений через запятую>};

Если количество значений элементов в наборе меньше, чем заявленное количество элементов, то в этом случае все элементы, не имеющие значений в указанном наборе будут иметь значение равное нулю.

Либо можно не указывать количество элементов массива (их количество будет определено из списка инициализации)

Тип_данных Имя_массива[]={<набор значений через запятую>};

Доступ к тому или иному значению элемента массива определяется следующим образом:

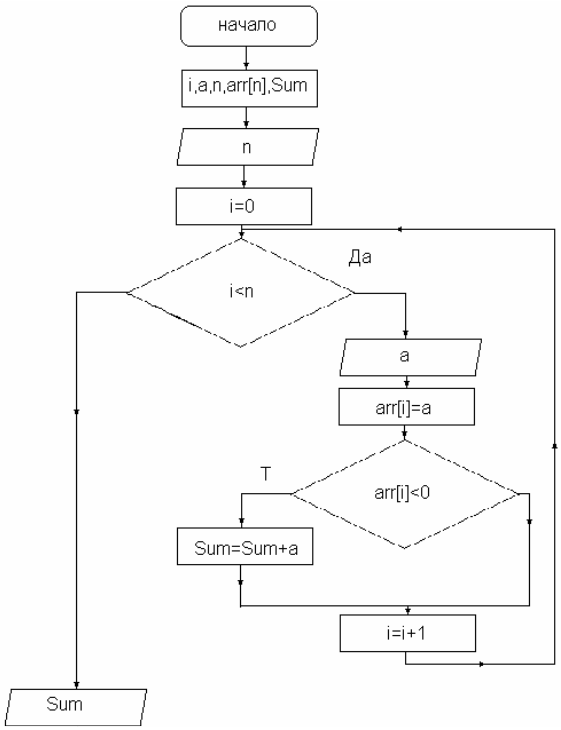
- сохранение значения в элементе массива

Имя_массива[*номер элемента*]=*значение*;

- присвоение значения элемента массива другой переменной

Имя_переменной=*Имя_массива*[*номер элемента*];

Пример. Вводится последовательность из *n* целых чисел. Сохранить все введенные значения и найти сумму всех отрицательных чисел.
Алгоритм решения задачи выглядит следующим образом:



Решение.

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
   system("chcp 1251");
   int Sum=0, a, n, arr[100];
   printf("Введите количество членов последовательности не более 100 ");
   scanf("%d",&n);
   for(int i=0; i<n;i++)
     {
       printf("Введите значение a[%d] =>",i+1);
       scanf("%d", &a);
      arr[i]=a;
                 if (arr[i]<0)   Sum+=arr[i];
     }
   printf("Значение суммы отрицательных членов последовательности %6d\n", Sum);
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Содержание отчета

1. Задание на выполнение.
2. Описание алгоритмов.
3. Результаты тестирования программы.

Порядок выполнения работы

Порядок выполнения работы можно представить следующим алгоритмом:

1. Ознакомиться с заданием.
2. Составить алгоритм решения задач.
3. Написать, отладить и протестировать программу .
4. Написать отчет.
5. Защитить работу.

Работа оценивается в **8** баллов

6 баллов – составление алгоритмов и написание программы
2 балла – отчет.

Лабораторная работа №4. *Динамические матрицы. Функции.*

Цель работы: изучение способов работы с динамическими двумерными массивами; реализация простых алгоритмов сортировки и поиска, оформление решения с помощью функций.

Инициализация матриц

Рассмотрим механизм инициализации динамической двумерной матрицы. Для работы с целочисленной матрицей в программе необходимо описать указатель на указатель:

*int **x;*

После этого необходимо выделить память под массив указателей на строки матрицы:

x=new int[n]; //* массив из указателей.

Каждый из *n* указателей должен указывать на массив из *m* элементов

```
for(int i=0;i<n;i++)
x[i] = new int [m]; //
```

массив из *m* элементов.

У матрицы заданной таким образом *n* строк и *m* столбцов.

Перед окончанием программы, работающей с динамической матрицей необходимо освободить используемую память. Освобождение памяти проходит так же в два этапа, но в обратном порядке:

```
for(i=n-1;i>=0;i--)
   delete [] x[i];
 delete [] x;
```

Для перебора элементов матрицы используются не один, а два цикла:

```
for(i=0;i<n;i++)
  for(j=0;j<m;j++)
        Обращение к элементу x[i][j]
```

Первый индекс элемента определяет номер строки, в которой он расположен, а второй – номер столбца.

Циклы, организованные выше, просматривают матрицу по строкам: элементы нулевой строки, элементы первой строки и т.д..

А такие циклы –

```
for(i=0;i<m;i++)
  for(j=0;j<n;j++)
        Обращение к элементу x[j][i]
```

просматривают элементы матрицы по столбцам, начиная с нулевого.

Задать элементы матрицы можно такими же способами, как и элементы массива: ввести с клавиатуры, задать случайным образом, рассчитать по формуле.

Следующий фрагмент программы задет элементы матрицы случайным образом (*n* – количество строк матрицы, *m* – количество столбцов):

```
…
float **x;
int n,m;
…
x = new float*[n];
for(int i=0;i<n;i++)
 x[i] = new float[m];
for(i=0;i<n;i++)
 for(int j=0;j<m;j++)
   x[i][j] = rand()%200/(rand()%100+1.);
…
```

Печать матриц

Для вывода элементов матрицы на экран так же используются два цикла. Организуем эти два цикла следующим образом: внешний цикл перебирает строки, внутренний - столбцы, и как только на экран выведены все элементы одной строки, вывод следующей строки нужно организовать с новой строки экрана:

```
// печать элементов вещественной матрицы x
for(i=0;i<n;i++)
 {
 for(int j=0;j<m;j++)
   printf("%8.3f ",x[i][j]);
// переход на новую строку экрана
 printf("\n");
 }
```

Синтаксис

Важным принципом структурного программирования является принцип модульности. В модульной программе отдельные части, предназначенные для решения частных задач, организованы в функции. Вы уже использовали стандартные функции вывода на экран, чтения с клавиатуры, и так далее. При такой организации, один и тот же фрагмент программы можно использовать несколько раз, не повторяя его текст. Еще одним преимуществом модульного программирования является легкость отладки, чтения и тестирования программы. Приведем пример задачи, которая требует использования модульного подхода к программированию: сформировать три целочисленных массива, элементами которых являются случайные числа – *A[5], B[10], C[25].* Для каждого массива найти сумму минимального и максимального элементов. Для решения этой задачи можно написать четыре функции – функцию создания массива, функцию печати массива, функции поиска минимального и максимального элементов. В основной функции *main* эти четыре функции будут использоваться для каждого из заданных массивов.

Синтаксически любая функция языка Си описывается следующим образом:

```
< тип возвращаемого результата> имя функции (<список формальных аргументов>)
{
  тело функции
```

```
}
```

Объявление и вызов функций

В языке Си объявить функцию можно несколькими способами. Первый способ – написать функцию до функции *main()*. Например, напишем функцию поиска минимального числа из трех заданных чисел.

```
#include <cstdlib>
#include <iostream>

using namespace std;
int min(int a1, int a2, int a3)
//  Тип возвращаемого результата – целый, формальные аргументы – три
// целых числа a1,a2,a3.
{
  int m = a1;
  if (m>a2) m = a2;
  if (m>a3) m = a3;
  return m; // возвращение результата с помощью оператора return }

int main(int argc, char *argv[])
{
    system("chcp 1251");
    int  x,y,z;
     printf("Введите значение x - ");
    scanf("%d",&x);
    printf("Введите значение y - ");
    scanf("%d",&y);
    printf("Введите значение z - ");
    scanf("%d",&z);
    int k = min(x,y,z); // вызов функции с реальными аргументами x,y,z.
    // переменной k присваивается возвращаемое значение
    printf("\n Минимальное значение - %d \n",k);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Итак, функцию можно описать до основной функции *main(),* передать результат в *main()* помогает оператор *return <возвращаемое значение>*. Оператор *return* кроме этого передает управление в вызывающую функцию. Таким образом, оператор может служить и для окончания работы функции. При вызове функции формальные аргументы заменяются реальными аргументами.

Механизм работы модульной программы может быть описан следующим образом:

- при компиляции программы, имеющей пользовательские функции, для каждой функции создается отдельный исполняемый код;
- при вызове функции выполнение программы прерывается, все данные программы сохраняются в стеке, начинает выполняться код тела функции, при этом происходит замена формальных аргументов на реальные, с которыми была вызвана функция;
- при достижении оператора *return* или закрывающей фигурной скобки функции управление передается в точку вызова вызывающей функции, при этом из стека возвращаются все сохраненные параметры.

Тело функции может быть описано и после тела вызывающей функции, или даже в другом файле, но в этом случае необходимо использовать прототипы функции. Прототипом функции называется указание типа возвращаемого результата, имени функции и списка типов формальных аргументов. Например, для функции *min* прототип будет выглядеть следующим образом:

```
int min(int,int,int);
```

При этом само тело функции может располагаться как после функции *main(),* так и в другом файле. Использование прототипов связано с принципом обязательного первоначального описания объектов, используемых в программе, то есть, как и любую переменную, функцию необходимо описать перед ее использованием.

Использование прототипов позволяет объединять функции в библиотеки. Для каждой библиотеки может быть написан заголовочный файл с расширением *h,* в котором будут перечислены прототипы всех функций библиотеки. Тела функций при этом могут храниться в отдельном файле с расширением *c (cpp).* Заголовочный файл подключается к файлу, содержащему вызывающую функцию с помощью директивы *include,* выполняется многофайловая компиляция программы (создание проекта). Например, следующий пример описывает небольшую библиотеку простейших арифметических функций.

Файл *ariphm.h*

```
int NOD(int,int); // функция нахождения наибольшего общего делителя //для двух целых чисел
float min(float,float);// функция нахождения минимального
                       //из двух вещественных чисел
float max(float,float);// функция нахождения максимального
                       // из двух вещественных чисел
```

Файл *ariphm.cpp*

```
// поиск минимального числа
// аргументы функции два вещественных числа x и y
// функция возвращает результат вещественного типа.
float min (float x, float y)
{
 if (x>y) return y;
          else return x;
}
// поиск максимального числа
// аргументы функции два вещественных числа x и y
// функция возвращает результат вещественного типа.
float max (float x, float y)
{
 if (x>y) return x;
          else return y;
}

// поиск наибольшего общего делителя
// аргументы функции два целых числа x и y
// функция возвращает результат целого типа.
int NOD(int x, int y) // поиск
{
  int z  = max(x,y);
  int l = min(x,y);
  int k = l;
  while(z%l!=0)
  {
   k = z%l;
   z = max(k,y);
   l = min(k,y);
  }
  return k;
}
```

Файл *Demo_A.cpp* – демонстрация вызова ранее описанных функций.

```
#include "ariphm.h" // подключение собственного заголовочного файла.
#include <cstdlib>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
   system("chcp 1251");printf("Введите два числа: ");
   int m,n;
   scanf("%d",&m);
   scanf("%d",&n);
  // Вызов функции NOD. Формальные аргументы x,y заменены
  //фактическими m и n.
   printf("Наибольший общий делитель: %d \n",NOD(n,m));
   // Вызов функции min. Формальные аргументы x,y заменены
   //фактическими m и n. Результат работы функции явно
// преобразован к типу int.
 printf("Минимальное число: %d \n",(int)min(n,m));
// Вызов функции min. Формальные аргументы x,y заменены
//фактическими m и n. Результат работы функции явно
// преобразован к типу int.
 printf("Максимальное число: %d \n",(int)max(n,m));
system("PAUSE");
return EXIT_SUCCESS;
}
```

Для того, чтобы данный пример выполнился необходимо использовать многофайловую компиляцию. В проект включаются файлы *Demo_A.cpp* и *ariphm.cpp.* Все файлы должны находиться в одной папке. Добавление файлов в проект – меню Project - Add to Project.

Локальные переменные

Переменные, описанные в теле функции, называются локальными переменными. Такие переменные видимы только внутри функции и создаются только при вызове функции. При достижении конца функции эти переменные уничтожаются, память, выделенная под хранение таких переменных, освобождается.

```
#include <stdio.h>
#include <conio.h>
 int x = 1;
void func1 (){
  int m = 12;
  printf("\n x = %d \n", x);
  printf("\n m = %d \n", m);}
int main(int argc, char *argv[])
{
   int m = 2;
  printf("\n x = %d \n", x);
  printf("\n m = %d \n", m);
  func1();
  printf("\n m = %d \n", m);
  system("PAUSE");
 return EXIT_SUCCESS;
}
```

Переменная *x*, по отношению к переменным *m*, объявленным в функциях *main()* и *func()* называется глобальной переменной. Зона ее видимости – весь файл, содержащий программу, Время существования – время работы функции *main()*.

Переменная *m* описанная в функции *func()* видна только внутри операторных скобок, ограничивающих тело функции.

Переменная *m*, описанная в функции *main()* видна только внутри операторных скобок, ограничивающих тело функции.

Результат работы программы будет следующим:

```
x = 1 // Выводится значение глобальной переменной x.
m = 2 // Выводится значение переменной m, описанной в
      //функции main()
// При вызове функции func() значение переменной m = 2
//сохраняется в стеке. Описывается новая
// переменная m, инициализируется значением
//12. На экран выводится значение переменной x, которая
//остается доступной и в функции.
x = 1
// Выводится значение m.
m = 12
// При завершении работы функции func(), переменная m
//перестает существовать. Из стека возвращается
// значение переменной m,  описанной в функции main().
m = 2.
```

Попробуйте изменить значение переменной *x* в функции *func()* и попытайтесь объяснить полученный результат.

Выход из функций

Выйти из функции (вернуться в вызывающую функцию, закончить выполнение функции) можно несколькими способами

При достижении закрывающей операторной скобки:

```
void PRINT(int x)
{
printf("Значение %d ->",x);
}
```

Функция *PRINT(int x)* закончит свое выполнение по достижению закрывающей скобки.

При достижении оператора *return*:

```
float func(float x)
{
 x=x*180/3.14;
return sin(x);
}
```

Функция *func(float x)* заканчивает свою работу при выполнении оператора *return* и передает в вызывающую функцию значение синуса заданного значения *x*.

```
void my_function(int x, int y){
if (y==0) {printf("деление на 0");return;}
float d  = ((float)x)/y;
printf("x/y = %f");
}
```

Функция закончит работу, если параметр *y* равен 0.

Порядок выполнения работы

1. Получить индивидуальный вариант.
2. Изучить теоретические аспекты лабораторной работы.

3. Разработать и реализовать на языке Си алгоритм решения предложенных задач с использованием функций.
4. Составить отчет по лабораторной работе.
5. Защитить работу.

Содержание отчета

1. Индивидуальное задание.
2. Описание алгоритмов решения задач.
3. Описание функций.
4. Результаты тестирования программы на матрице [5x5].

Лабораторная работа №5. *Строки*

Цель работы: реализация алгоритмов работы со строками, изучение стандартных функций для работы со строками в языке Си.

Инициализация строк

Язык Си имеет очень мощный механизм для работы со строками. Специального типа для представления строковых данных в Си нет, строка – это массив символов. Для описания строк можно использовать массив символов с постоянной длиной:

char str[25]; // строка из 24 символов.

Или массив символов с динамическим выделением памяти:

*char *str = new char[25]; //* строка из 24 символов.

Задать строку данных можно различными способами:

- задать с помощью строковой константы: *char str[25] = "Пример строки";*
- прочитать с клавиатуры с помощью функции *scanf(): scanf("%s",str);* обратите внимание, строка символов – массив, а любой массив в Си - это адрес, поэтому перед переменной *str* не ставится знак *&;* функция *scanf()* читает строку до первого встреченного пробела; остальная часть строки обрезается;
- прочитать с клавиатуры с помощью функции *gets() – str = gets(str);* в случае успешного чтения функция возвращает прочитанную строку, в противном случае – *null;* функция может читать произвольную строку, однако надо помнить, что с клавиатуры можно ввести не более 160-ти символов.
- задать строку посимвольно – *str[i] = 'A';*

8.3.2. Представление строки в памяти компьютера

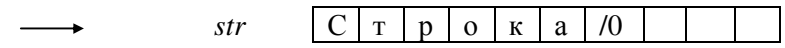
Строка Си может состоять из произвольного количества символов. Окончанием любой строки считается так называемый символ «нуль-терминатор» - '/0'.

Пусть в программе выполнены следующие действия:

char S = new char[10];*

S = "Строка";

Представление такой строки в памяти выглядит следующим образом:



После '/0' в памяти может быть расположены любые символы, но каждое обращение к строке подобно просмотру памяти, начиная с адреса *str,* пока не будет встречен '/0'. Символ '/0' считается при выделении памяти под строку. Поэтому строка *str* может содержать только 9 символов. Если выполнена попытка записать в выделенную память более, чем 9 символов, не произойдет ошибки выполнения, но программа может работать некорректно, если «лишние» символы «испортят» другие данные программы. Такие ошибки работы со строками наиболее часты.

Стандартные функции считывания строки автоматически добавляют символ окончания строки. Если же Вы формируете строку посимвольно, не забудьте записать в последнюю позицию строки символ ‘/0’.

При работе со строками всегда нужно помнить, что имя строки – указатель, поэтому присваивание типа:

*char *my = new char[10];*
char z = new char[20];*
my = “Hello”;
z = “world”;
my = z;

приведет к тому, что и адрес *z* и адрес *my* будут указывать на строку *“world”*. Если Вы хотите, чтобы в памяти появилась две копии строки *“world”,* необходимо выполнить копирование строки с помощью функции *strcpy().*

В Си нельзя сравнивать и складывать строки с помощью стандартных арифметических операций. Для этих действий так же используются специальные функции.

Стандартные функции для работы со строками

Прототипы ниже описанных функций находятся в заголовочном файле *string.h.*

*char *strcat(char *dest, const char *src);* - присоединяет копию строки *src* к концу строки *dest*. Длина полученной строки равна *strlen(dest) + strlen(src)*. Возвращает указатель на объединенную строку.

*char *strchr(const char *s, int c);*- просматривает строку *s* в прямом направлении в поисках заданного символа *c*. Ищет *первое* вхождение символа *c* в строку s. Возвращает указатель на первое вхождение символа *c* в строку *s*, если *c* не входит в строку *s* функция возвращает *null*.

*int strcmp(const char *s1, const char *s2);* - производит беззнаковое сравнение символов строк *s1* и *s2*, начиная с первого символа каждой строки и продолжая то же с последующими, пока не встретятся два соответствующих не совпадающих символа, или не будет достигнут конец данных строк. Возвращает следующия значения: отрицательное число , если *s1* меньше *s2*; 0, если *s1* совпадает с *s2;* положительное число, если *s1* больше *s2.*

*int strcmpi(const char *s1, const char *s2);* - выполняет беззнаковое сравнение *s1* и *s2*. Возвращает следующия значения: отрицательное число , если *s1* меньше *s2*; 0, если *s1* совпадает с *s2;* положительное число, если *s1* больше *s2.*

*char *strcpy(char *dest, const char *src);*- копирует строку *src* в строку *dest*. Копирование завершается после достижения терминального нуль-символа. Возвращает *dest*.

*size_t strcspn(const char *s1, const char *s2);* - возвращает длину начального сегмента строки *s1*, который полностью состоит из символов, не встречающихся в *s2*.

*size_t strlen(const char *s);*- вычисляет длину строки *s.* Возвращает количество символов в строке *s,* не считая терминального нуль-символа.

*char *strlwr(char *s);* - преобразует прописные буквы (от *A до Z)* в строчные *(от a до z).* Никакие другие символы не изменяются. Возвращает указатель на строку *s.*

*char *strncat(char *dest, const char *src,*
size_t maxlen); - копирует не более *maxlen* символов из *src* в конец *dest* и добавляет терминальный нуль-символ. Максимальная длина полученной строки равна *strlen(dest) + maxlen.* Возвращает указатель на *dest.*

*int strncmp(const char *s1, const char *s2,*
*size_t maxlen);*выполняет беззнаковое сравнение, проверяя не более *maxlen* символов. Она начинает с первого символа каждой строки и продолжает то же с последующими, пока не встретятся два соответствующих не совпадающих символа, или не будет проверено *maxlen* символов. Возвращает следующия значения: отрицательное число , если *s1* меньше *s2*; 0, если *s1* совпадает с *s2;* положительное число, если *s1* больше *s2.*

*char *strncpy(char *dest, const char *src,*
size_t maxlen); - копирует не более *maxlen* символов из *src* в *dest,* усекая *dest* или дополняя нуль-символами. Строка назначения *dest* может не заканчиваться нуль-символом, если длина *src* больше или равна *maxlen.* Возвращает указатель на *dest.*

*char *strnset(char *s, int ch, size_t n);-* помещает символ *ch* в первые *n* байтов строки *s.* Если *n > strlen(s),* вместо *n* берется значение *strlen(s).* Функция завершается, если *n* символов уже заполнены, или обнаружен терминальный нуль-символ. Возвращает *s.*

*char *strpbrk(const char *s1, const char *s2); -* просматривает строку *s1,* пока не встретит вхождение любого символа из *s2.* Возвращает указатель на первое вхождение любого из символов, содержащихся в *s2.* Если ни один из символов *s2* не содержится в *s1,* функция возвращает *null.*

*char *strrchr(const char *s, int c); -* в поисках указанного символа *strrchr* просматривает строку в обратном направлении. Эта функция ищет последнее вхождение символа с в строку *s.* Возвращает указатель на последнее вхождение символа *c.* Если *c* не содержится в *s,* возвращается *null.*

*char *strrev(char *s); -* изменяет порядок следования символов в строке на обратный, за исключением терминального нуль-символа. Возвращает указатель на реверсированную строку.

*char *strset(char *s, int ch);-* заполняет всю строку *s* символом *ch.* Она завершает работу при достижении терминального нуль-символа. Возвращает указатель на строку *s.*

*size_t strspn(const char *s1, const char *s2); - strspn* ищет начальную подстроку строки *s1*, целиком состоящую из символов, содержащихся в строке *s2.* Возвращает длину начальной подстроки *s1,* состоящей только из символов, содержащихся в строке *s2.*

*char *strstr(const char *s1, const char *s2); -* просматривает строку *s1* до первого обнаружения вхождения подстроки *s2.* Возвращает указатель на первый символ первого вхождения *s2* в *s1.* Если подстроки *s2* в *s1* не содержится, strstr возвращает *null.*

*double strtod(const char *s, char **endptr); -* преобразует символьную строку *s* в значение типа *double. s* есть последовательность символов, которая может быть интерпретирована как значение типа *double;* она должна иметь следующий обобщенный формат:

[ws] [sn] [ddd] [.] [ddd] [fmt [sn] ddd]

- *[ws]* - необязательные пробелы
- *[sn]* - необязательный знак (+ или -)
- *[ddd]* - необязательные цифры
- *[fmt]* - необязательный символ e или E
- *[.]* - необязательная десятичная точка

Функция прекращает работу при обнаружении первого символа, который не может быть интерпретирован как соответствующая часть значения типа *double.* Возвращает значение типа *double,* полученное из строки *s.*

*char *strtok(char *s1, const char *s2); -* предполагает, что строка *s1* состоит из последовательности из нуля или более лексических единиц, разделенных одним или несколькими символами из строки разделителей *s2.* Первое обращение к *strtok* возвращает указатель на первый символ первой лексической единицы в *s1* и записывает нуль-символ в *s1* непосредственно за этой лексической единицей. Последующие обращения с первым аргументом, установленным в нуль, будут продолжать просматривать строку *s1,* пока не исчерпаются все содержащиеся в ней лексические единицы. Возвращает указатель на лексическую единицу, обнаруженную в *s1.* Нулевой указатель возвращается, если таковых больше нет.

*long strtol(const char *s, char **endptr, int radix); -* преобразует символьную строку *s* в значение типа *long.* Строка *s* есть последовательность символов, которая может быть интерпретирована как значение типа *long;* она должна иметь следующий обобщенный формат:

[ws] [sn] [0] [x] [ddd]

- *[ws]* - необязательные пробелы
- *[sn]* - необязательный знак (+ или -)
- *[0]* - необязательный ноль (0)
- *[x]* - необязательный символ *x* или *X*
- *[ddd]* - необязательные цифры

Возвращает значение преобразованной строки или 0 в случае ошибки.

*unsigned long strtoul(const char *s, char **endptr, int radix);* - работает так же, как и *strtol,* за исключением того, что строка *s* преобразуется в значение типа *unsigned long,* в то время, как *strtol* преобразует в значение типа *long.* Возвращает преобразованное значение типа *unsigned long или 0* в случае ошибки.

*char *strupr(char *s);-* преобразует строчные буквы *(a-z),* содержащиеся в строке *s,* в прописные *(A-Z).* Никакие другие символы не изменяются. Возвращает указатель на строку *s.*

Примеры решений задач со строками

Рассмотрим пример, формирующий строку *words* по следующему правилу – дана произвольная строка символов *dest*, в строку *words* записать все первые символы слов строки. Словом считается последовательность символов, ограниченная пробелами или знаками препинания и не имеющая пробелов внутри себя.

```
int main(int argc, char *argv[])
{
   system("chcp 1251");
   char dest[200];
   printf("Введите строку символов: ");
   // Ввод строки
   gets(dest);
   // Используем для работы вспомогательную строку buf.
   // Выделение памяти под строку
   char *buf = new char [strlen(dest)+1];
   // Копирование строки  dest в строку buf
   strcpy(buf,dest);
   char *words;
   // Подсчет количества слов
   int k = 0;
   // Выделение первого слова
   char *temp = strtok(buf," ,;.!?-");
  // Выделение последующих слов
   while (temp!=NULL)
    {
      temp = strtok(NULL," ,;.!?-");
      k++;
     }
   // Если строка содержит хотя бы одно слово
   if (k){
    // Выделить память под строку, в которой будут храниться
   // первые символы слов
   words = new char[k+1];
   k = 0;
   // провести выделение слов вновь
   temp = strtok(dest," ,;.!?-");
   // сохраняя первый символ каждого выделенного слова
   // в строке  words.
      words[k++] = temp[0];
     while (temp!=NULL)
      {
        temp = strtok(NULL," ,;.!?-");
        if (temp)
             words[k++] = temp[0];
       }
// последним символом строки записать
// нуль-терминатор
 words[k] = '\0';
// вывести строку на экран
 puts(words);
 }
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

Дана произвольная строка символов, найти слово с максимальной длиной и вывести его на экран.

```
int main(int argc, char *argv[])
{
   system("chcp 1251");
   char dest[200];
   printf("Вводите строку: ");
   gets(dest);
   char *buf = new char [strlen(dest)+1];
   strcpy(buf,dest);
   // Выделить первое слово строки
   char *temp = strtok(buf," ,;.!?-");
   // Принять это слово за слово с максимальной длиной
.  // Выделять последующие слова и сравнивать их со
  // словом с максимальной длиной
   int max = strlen(temp);
   char *strmax = new char[max+1];
   strcpy(strmax,temp);
.   while (1)
   {
       temp = strtok(NULL," ,;.!?-");
       if (!temp)break;
       if (max<strlen(temp))
                       {
                       delete [] strmax;
                       max = strlen(temp);
                       strmax = new char[max+1];
                       strcpy(strmax,temp);
                        } }
     printf("В строке // %s //\n слово с максимальной длиной // %s //",dest,strmax);
    printf("\n Его длина - %d",max);
   delete [] strmax;
   delete [] buf;
   system("PAUSE");
   return EXIT_SUCCESS;
}
```

В произвольной строке символов найти количество символов, не являющихся буквами.

Для решения этой задачи воспользуемся функцией *isalpha(),* которая передает ненулевое значение, если проверяемый символ является буквой и нуль в противном случае. Прототип функции описан в заголовочном файле *ctype.h*

```
int main(int argc, char *argv[])
{
   system("chcp 1251");
   char dest[200];
   printf("Вводите строку символов: ");
   gets(dest);
   int k = 0;
   for(int i=0;i<strlen(dest);i++)
         {
          if(isalpha(dest[i])==0) k++;
```

```
        }
        printf("Количество символов не являющихся буквами  - %d\n",k);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

Порядок выполнения работы

1. Получить индивидуальный вариант
2. Ознакомиться со стандартными функциями Си для работы со строками.
3. Разработать и реализовать на языке Си алгоритм по индивидуальному варианту
4. Составить отчет по лабораторной работе
5. Защитить работу.

Содержание отчета

1. Индивидуальное задание.
2. Описание алгоритма.
3. Результаты тестирования программы.

Лабораторная работа №6. *Текстовые файлы*

Цель работы – получить навыки работы с текстовыми файлами.

Типы файлов в Си

Библиотечные функции для работы с файлами можно разделить на две группы – **префиксные и потоковые.** Для каждой открытой программы операционная система формирует уникальную таблицу открытых файлов, каждый файл имеет свой собственный номер – префикс. Первые 4 префикса зарезервированы под устройства стандартного ввода-вывода – ***stdin*** **-** клавиатура**,** ***stdout*** **-** экран**,** ***stderr*** **-** экран**,** ***stdaux*** - порт COM1, ***stdprn*** **-** принтер**.** Если в программе открывается файл, то операционная система автоматически присваивает ему первый свободный номер, этот номер будет префиксом файла.

Для каждой из групп файлов установлены два режима доступа к файлу – двоичный и текстовый. При текстовом режиме доступа символы 0DH 0AH (перевод каретки) преобразуются в один символ '\n', соответственно при записи символ перевода каретки преобразуется в пару символов. При считывании из файла в текстовом режиме чтение символа 1AH заканчивает считывание информации из файла (символ конца файла).

При двоичном доступе к файлу каждый символ считывается отдельно, как не имеющий ни какого смысла. Режим доступа к файлу задается непосредственно при использовании библиотечной функции открытия или через переменную *_fmode (stdio.h),* которая по умолчанию установлена в *O_TEXT*, для двоичного доступа - *O_BINARY*.

Функции поточного ввода-вывода называют стандартными функциями ввода-вывода. Си создает внутреннюю структурную переменную по шаблону *FILE* (описание находится в *stdio.h*).

Механизм чтения-записи

Файл – это поименованная область на жестком диске, заполненная какой-либо информацией. В конце каждого файла записывается символ окончания файла, который помогает операционной системе корректно работать с файлами.

Различают понятия «открыть файл для записи» и «открыть файл для чтения». При первом способе открытия файла вся имеющаяся в нем информация стирается, и Вы можете записать него новую информацию. При открытии файла создается переменная, которая содержит в себе адрес памяти, где записан просматриваемый файл. При записи этот указатель меняет свое положение – передвигается по файлу. Как только произошла запись, указатель содержит адрес, по которому возможно, произойдет следующая запись.

Аналогичен и механизм чтения информации из файла. В этом случае указатель переходит к следующей позиции после считывания информации. Таким образом, если произошло чтение символа, то указатель передвинется в файле на 1 байт, если Вы считываете строку – на размер строки и т.д. В двоичном файле передвижение проходит на размер считываемого элемента. Например, при считывании целочисленных данных указатель чтения-записи передвигается на два байта, при считывании данных типа *float* – на четыре байта.

Функции для поточного доступа к файлам

FILE fopen (const char *filename, const char* mode);* -возвращает указатель на переменную типа *FILE*, *mode* – заданный режим открытия файла. При неуспешной работе функция возвращает *NULL*. Во избежание ошибок при открытии файла необходимо проверять результат выполнения функции, например, как в следующем фрагменте программы:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
 {
 FILE *f;
 char name[] = "prim.txt";
```

```
clrscr();
if ((f = fopen(name,"rb"))==NULL) {
  printf( "Ошибка открытия файла: ");
getch();
  }
  else…
…
}
```

Обратите внимание: очень распространенная ошибка – при указании полного имени файла символы '\' строке имени файла должны удваиваться (например: ***n:\\c++\\file.txt***).

Если Вы не указываете полное имя, то файл ищется (создается) в текущей директории.

Режимы доступа к файлу:

r - открыть для чтения.

w - создать для записи. Если файл с таким именем уже существует, он будет перезаписан.

a – открыть файл для обновления, открывает файл для записи в конец файла или создает файл для записи, если файла не существует.

r+- открыть существующий файл для корректировки (чтения и записи).

w+- создать новый файл для корректировки (чтения и записи). Если файл с таким именем уже существует, он будет перезаписан.

a+ - открыть для обновления; открывает для корректировки (чтения и записи) в конец файла, или создает, если файла не существует.

t – открыть файл в текстовом режиме.

b– открыть файл в двоичном режиме.

Два эти аргумента указываются вторыми символами в строковой переменной *mode,* например "*r+b*".

По умолчанию установлен текстовый режим доступа к файлу.

*int fclose(FILE *fp)* - закрывает файл *fp,* при успешной работе возвращает 0, при неуспешной *EOF).*

int closeall(void) – закрывает все файлы, открытые в программе, при успешной работе возвращает число закрытых потоков, при неуспешной - *EOF.*

*FILE *freopen(const char *filename, const char* mode, FILE *stream)* – закрывает поток *stream*, открывает поток *filename* с новыми правами доступа установленными в *mode*. Если потоки разные, то происходит переадресация потока *stream* в поток *filename.*

```
void main()
 {
 char name[] = "prim.txt";
 int n;
 FILE *f;
  printf( "Введите переменную n: ");
scanf("%d", & n);
freopen(name,"wt",stdout);
// переопределить стандартное устройство вывода - экран
  for (int i=0;i<n;i++)
   printf("%d\n",i);
// печать будет осуществляться в текстовый файл с именем
// prim.txt
printf("Press any button...");
getch();
 }
```

ch = fgetc(<указатель на файл>) – возвращает символ *ch* из файла, с которым связан указатель.

ch = getc(<указатель на файл>) – возвращает символ *ch* из файла, с которым связан указатель.

fputc(ch,<указатель на файл>) – записать символ *ch* в указанный файл.

putc(ch,<указатель на файл>) – записать символа *ch* в файл.

fgets(str, n, <указатель на файл>) **–** прочитать строку *str*, длиной *n* символов, или до первого встреченного \n из файла.

fputs(str, <указатель на файл>) – записать строку *str*, в файл. Символ перевода на другую строку в файл не записывается.

fscanf(<указатель на файл>, управляющая строка, ссылка) – универсальная функция считывания из текстового файла. Например, *fscanf(f,"%d",&n)* считатывает из файла *f* целое число в переменную *n*.

fread(ptr,size,n,<указатель на файл>) **–** считывает *n* элементов размером *size* в область памяти, начиная с *ptr*. В случае успеха возвращает количество считанных элементов, в случае неуспеха – *EOF*.

fwrite(ptr,size,n,<указатель на файл>) – записывает *n* элементов размером *size* из памяти, начиная с *ptr* в файл– в случае успеха возвращает количество записанных элементов, в случае неуспеха – *EOF*.

fprintf((<указатель на файл>, управляющая строка, [список аргументов]) – форматированный вывод в файл.

В Си к любому файлу может быть осуществлен прямой доступ. Для прямого доступа используются следующие функции:

rewind(<указатель файла>) **–** установить указатель файла на начало файла.

int fseek(<указатель файла>, offset, fromwheare) - установить указатель чтения-записи файла на позицию *offset,* относительно позиции *fromwheare. fromwheare* может принимать значения SEEK_END – от конца файла, SEEK_SET – от начала файла, SEEK_CUR – от текущей позиции.

long int n = ftell(<указатель на файл>) – в переменную *n* передать номер текущей позиции в файле.

int z = fgetpos(<указатель файла>, npos); в динамической памяти по адресу *npos* записать номер текущей позиции в файле, в случае успеха функция возвращает 0; в противном случае – любое ненулевое число.

int unlink(<имя файла>) – удаление файла, при успехе функция возвращает 0, при неуспехе - -1.

int rename(<старое имя>,<новое имя>) – переименованиие файла, при успехе функция возвращает 0, при неуспехе - -1.

int feof(<указатель на файл>) возвращает 0, если конец файла не достигнут, любое ненулевое число, если достигнут.

int ferror(<указатель на файл>) возвращает ненулевое значение, если при работе с файлом возникла ошибка, 0 – в противном случае.

В Си описана группа функций для управления работы с файлами – проверка на существование файла или директория, поиск файлов и др.

Примеры работы с текстовыми файлами

Запись данных в текстовый файл

В текстовый файл можно записать данные различных типов. В дальнейшем, содержимое такого файла можно просмотреть в любом текстовом редакторе.

Пример 1. Создать вещественный массив случайным образом и сохранить его в текстовом файле.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
 {
 clrscr();
 char name[25];
 int n;
 FILE *f;
 int flag = 1;
// Создадим цикл, позволяющий корректировать ввод
// имени файла
```

```
 do
 {
 printf( "Введите имя создаваемого файла: ");
 scanf("%s",name);
// Попытка открыть файл для чтения. Если такой файл уже
// существует, то задать вопрос пользователю
 if ((f = fopen(name,"r"))!=NULL)
  {
// Заменить существующий файл?
   printf("Файл уже существует. Заменить? (y/n)");
   char ch = getch();
// Если пользователь нажал кнопку «n», очистить экран,
// вернуться к началу цикла
   if (ch == 'n') {clrscr(); continue;}
  }
// В этот блок программы управление попадет только если
// пользователь подтвердил замену или задал имя
// несуществующего файла.
// Создать файл.
   if ((f=fopen(name,"w"))==NULL)
    {
     printf("Ошибка создания файла");
     getch();
     break;
    }
   printf("\nВведите размерность массива: ");
   scanf("%d",&n);
   for(int i=0;i<n;i++ )
    { float y = random(100)/(random(50)+1.)-random(30);
// На одной строке файла печатать только 10 элементов.
      if (i>=10&&i%10==0) fprintf(f,"\n");
     fprintf(f,"%8.3f",y);
    }
// Закрыть файл.
   fclose(f);
// Закончить цикл
   flag = 0;
  } while (flag);
  printf("Файл создан. Для окончания работы нажмите
любую клавишу...");
 getch();
 }
```

Обратите внимание: в программе не использовался массив. Т.к. данные сохраняются в файле, для решения задачи достаточно одной целочисленной переменной.

Чтение данных из текстового файла

Предположим, в текущем каталоге существует файл *data.txt.*

Пример 1. На первой строке в файле записана размерность целочисленной матрицы. Далее – сама матрица. Считать матрицу в память и вывести ее на экран. Данные записаны в файле *my.txt.* Записать такой файл можно, например, в блокноте или любом другом текстовом редакторе. Если не указан полный путь к файлу, то файл должен находиться в текущей папке (той, из которой Вы запускали редактор Си).

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
```

```
void main()
{
FILE *f;
clrscr();
f = fopen("my.txt", "r");
// Проверка ошибки открытия файла
if (f==NULL) {
printf("Файл не найден... /n Для окончания работы нажмите любую клавишу...");
getch();
exit(0);
}
int n,m;
// Чтение размерности матрицы
fscanf(f,"%d",&n);
fscanf(f,"%d",&m);
int **a;
// Выделение памяти под матрицу
a = new int* [n];
for(int i=0;i<n;i++)
 a[i] = new int [m];
// Чтение матрицы
for(i=0;i<n;i++)
  for(int j=0;j<m;j++)
    fscanf(f,"%d",&a[i][j]);
printf("Прочитана матрица: \n");
// Печать элементов матрицы
for(i=0;i<n;i++)
{
  for(int j=0;j<m;j++)
     printf("%5d",a[i][j]);
  printf("\n");
}
getch();
}
```

В предыдущем примере размерность матрицы считывалась из файла. Но можно организовать считывание элементов и без заданной размерности. В этом случае файл сканируется по условию пока не найден конец файла, одновременно ведется подсчет считанных элементов.

Пример 2. В текстовом файле записано произвольное количество чисел. Считать данные из файла в массив и вывести на экран. Файл *my.txt* находится в текущей папке.

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
void main()
{
FILE *f;
clrscr();
f = fopen("my.txt", "r");
// Проверка ошибки открытия файла
if (f==NULL) {
printf("Файл не найден... /n Для окончания работы нажмите любую клавишу...");
getch();
exit(0);
}
int n=0,y;
int *a;
```

```
// пока не конец файла f
while (!feof(f))
{
// читать элемент и
 fscanf(f,"%d",&y);
// увеличивать счетчик.
 n++;
}
// После окончания цикла в переменной n хранится
// количество целых чисел, записанных в файле.
// Выделить память под массив.
a = new int [n];
// Указатель чтения-записи файла передвинуть в начало.
fseek(f,0,SEEK_SET);
// Читать n целых чисел из файла в массив.
for(int i=0;i<n;i++)
   fscanf(f,"%d",&a[i]);
printf("Прочитан массив: \n");
for(i=0;i<n;i++)
    printf("%5d",a[i]);
getch();
}
```

Изменение текстового файла

При решении некоторых задач не требуется считывать все данные из файла в оперативную память. Си позволяет выполнять изменения непосредственно в файле, используя механизм прямого доступа.

Пример 1. В текстовом файле расположен произвольный текст. Не считывая весь текст в память изменить все первые буквы слов на прописные.

Для решения этой задачи будем использовать свойство функции *fscanf* – функция читает строки до первого встреченного пробела. Поэтому, если организовать цикл по условию пока не найден конец файла и в теле цикла использовать функцию *scanf* для чтения строковых данных, то на каждом шаге цикла будет считываться ровно одно слово.

В прочитанном слове изменим первый символ на прописной, используя функцию *toupper(char ch),* функция преобразует символ *ch* в прописной, если это возможно. Результат работы функции – измененный символ.

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

void main()
{
clrscr();
// Откроем файл для чтения с дополнением.
// Символ окончания файла в этом случае автоматически
// удаляется.
FILE *f = fopen("text.txt","r+");
if (f==NULL) {
        printf("Файл не найден. \n");
        printf("Для окончания работы нажмите любую клавишу.\n");
        getch();
        exit(1);
        }
```

```
char word[100];
long pos1;
// Организуем бесконечный цикл для чтения файла.
//
while (1){
// Если прошло неуспешно, значит достигнут конец файла,
// в этом случае нужно закончить выполнение цикла
 if ( fscanf(f,"%s",word)!=1) break;
 // В переменную pos1 получить текущую
// позицию указателя чтения-записи.
pos1 = ftell(f);
// Установить указатель на позицию, с которой было
// считано слово.
 fseek(f,pos1-strlen(word),SEEK_SET);
 word[0] = toupper(word[0]);
 printf(" %s\n",word);
// Записать в файл измененное слово.
 fprintf(f,"%s",word);
// Установить указатель чтения-записи на позицию,
// находящуюся после измененного слова.
 fseek(f,pos1,SEEK_SET);

 }
fclose(f);
 }
```

Порядок выполнения работы

1. Получить индивидуальное задание
2. Составить и записать алгоритм решения задачи
3. Составить программу, реализующую алгоритм.
4.Выполнить компиляцию проекта
5. Написать отчет о проделанной работе.
6. Защитить работу

Задание оценивается в 5 баллов: 1 балл алгоритм,
1 балл код программы,
1 балла отчет,
2 балла защита.

Содержание отчета:

1.Текст задания
3. Алгоритм
4. Тестирование программы

Лабораторная работа №7. *Двоичные файлы*

Цель работы – научиться создавать, читать и обрабатывать двоичные файлы.

Запись и чтение информации в двоичный файл.

Рассмотрим сохранение и последующее чтение числовой информации в двоичном представлении.

Пример. Записать в двоичный файл *n* вещественных чисел, прочитать созданный файл и вывести на экран в виде матрицы с числом столбцов *m*. Решение оформить в виде функций.

Напишем функцию, создающую двоичный файл. Функция не будет возвращать значений, параметрами функции будут имя создаваемого файла и количество записываемых чисел:

*void create_file(char * name, int n);*

Функция чтения так же не будет возвращать значений, а параметрами функции будут имя читаемого файла и количество столбцов выводимой информации:

void read_file(char name, int m);*

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
void create_file(char *name, int n)
{
  FILE *f = fopen(name,"wb");
  if (f==NULL) {
            printf("Ошибка создания файла. \n");
            printf("Для окончания работы нажмите любую клавишу.\n");
            getch();
            exit(1);
            }
// Запись n чисел в файл
  for(int i=0;i<n;i++)
  {
   float z = random(200)/(random(100)+1.)-random(70);
   fwrite(&z,sizeof(float),1,f);
   }
  fclose(f);
}
void read_file(char *name, int m)
{
  FILE *f = fopen(name,"rb");
  if (f==NULL) {
            printf("Файл не найден. \n");
            printf("Для окончания работы нажмите любую клавишу.\n");
            getch();
            exit(1);
            }
//  Переменная для подсчета количества уже выведенных
//  значений.
  int i = 0;
  float z;
// Пока не конец файла
  while(!feof(f))
  {
// если количество выведенных элементов делится без
// остатка на заданное количество столбцов,
```

```
// перейти на следующую строку экрана.
   if (i>=m&&i%m==0)
      printf("\n");
   if(fread(&z,sizeof(float),1,f)!=1) break;
   printf("%8.3f",z);
   i++;
   }
  fclose(f);
}
void main()
{
clrscr();
char Fname[30];
int n,m;
printf("Введите имя создаваемого файла: ");
scanf("%s",Fname);
printf("Введите количество записываемых чисел: ");
scanf("%d",&n);
printf("Введите количество столбцов: ");
scanf("%d",&m);
create_file(Fname,n);
read_file(Fname,m);
getch();
}
```

Обратите внимание: значение функции *feof* формируется только при попытке чтения из файла. Поэтому, внутри цикла выполнена проверка значения функции *fread:*

if(fread(&z,sizeof(float),1,f)!=1) break;

Если чтение на данном шаге проведено неуспешно (найден конец файла), то закончить работу цикла.

Реализация прямого доступа в двоичном файле

В некоторых задачах требуется читать только указанную информацию. Механизм работы с файлами в Си позволяет обращаться к элементам, записанным на заданных позициях файла, без чтения предыдущих элементов.

Пример 1. В двоичном файле сохранена следующая информация – размерность *n* квадратной матрицы и сама вещественная матрица. Вывести на экран элементы заданного столбца *k.*

В двоичном файле элементы матрицы сохранены следующим образом:

x[0][0],x[0][1],…x[0][n],x[1][0],…,x[n-1][n-2],x[n-1][n-1].

Очевидно, что позиция *k*-того элемента рассчитывается по формуле:

*k*sizeof(*элемента*)+i*sizeof(*элемента*),*

где *i* – номер строки.

На рис. 10.1 представлено расположение в двоичном файле целочисленной матрицы с *n = 5.* Нумерация позиций в файле начинается с нуля. Число типа *int* занимает в памяти 2 байта. Поэтому, элемент матрицы, находящийся в нулевом столбце и нулевой строке сохранен в файле с позиции с номером 0, элемент с индексами 0 и 1 – с позиции 2, элемент с индексами *i* и *j* - с позиции *j*2+i*2.*

0	*2*	*4*	*6*	*8*
0	*2*	*4*	*6*	*8*
0	*2*	*4*	*6*	*8*
0	*2*	*4*	*6*	*8*

Рис. 10.1. Расположение элементов в двоичном файле

Воспользуемся этой закономерностью для решения задачи:

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
// Создание двоичного файла с именем name,
// содержащем n*n вещественных чисел.
void create_file(char *name, int n)
{
  FILE *f = fopen(name,"wb");
  if (f==NULL) {
              printf("Ошибка создания файла. \n");
              printf("Для окончания работы нажмите любую клавишу.\n");
              getch();
              exit(1);
              }
// Запишем в файл переменную n.
  fwrite(&n,sizeof(n),1,f);
// Запишем в файл элементы квадратной матрицы.
  for(int i=0;i<n*n;i++)
  {
   float z = random(200)/(random(100)+1.)-random(70);
   fwrite(&z,sizeof(float),1,f);
   }
  fclose(f);
}
// Функция чтения файла с именем name.
void read_file(char *name)
{
  int m;
  FILE *f = fopen(name,"rb");
  if (f==NULL) {
              printf("Файл не найден. \n");
              printf("Для окончания работы нажмите любую клавишу.\n");
              getch();
              exit(1);
              }
  int i = 0;
// Чтение размерности матрицы.
  float z;
  fread(&m,sizeof(int),1,f);
 // Чтение всей матрицы и вывод ее на экран.
 while(!feof(f))
  {
   if (i>=m&&i%m==0)
      printf("\n");
   if(fread(&z,sizeof(float),1,f)!=1) break;
   printf("%8.3f",z);
   i++;
   }
  fclose(f);
```

```
}
// Чтение элементов k-того столбца
void read_k(char *name, int k)
{
  int m;
  FILE *f = fopen(name,"rb");
  if (f==NULL) {
              printf("Файл не найден. \n");
              printf("Для окончания работы нажмите любую клавишу.\n");
              getch();
              exit(1);
              }
  float z;
  fread(&m,sizeof(int),1,f);
  for(int i=0;i<m;i++)
  {
    int l =i*m*sizeof(float) + k*sizeof(float)+sizeof(int);
    fseek(f, l,SEEK_SET);
    fread(&z,sizeof(z),1,f);
    printf("%8.3f",z);
   }
   fclose(f);

}
void main()
{
clrscr();
char Fname[30];
int n,m;
printf("Введите имя создаваемого файла: ");
scanf("%s",Fname);
printf("Введите количество строк матрицы: ");
scanf("%d",&n);
printf("Введите номер столбца: ");
scanf("%d",&m);
create_file(Fname,n);
printf("Матрица: \n");
read_file(Fname);
printf("Элементы столбца с номером %d \n",m);
read_k(Fname,m);
getch();
}
```

Аналогичным образом решаются задачи, требующие изменения уже существующих файлов.

Пример 2. В двоичном файле записано произвольное количество целых чисел. Не считывая все содержимое файла в память поменять порядок элементов на обратный – поменять местами первый и последний элементы, второй и предпоследний и т.д..

Будем считать, что файл уже создан с помощью функций, подобных функциям из предыдущих примеров.

Напишем функции печати заданного двоичного файла и изменения заданного файла.

```
#include <stdio.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
// Функция чтения файла, функция возвращает количество
// считанных данных.
```

```
int RP_f(char *name){
FILE *f = fopen(name,"rb");

int z;
int n = 0;
while(!feof(f))
{
 if(fread(&z,sizeof(z),1,f)!=1) break;
 printf("%4d",z);
 n++;
}
printf("\n");
fclose(f);
return n;
}
// Функция изменения файла. Параметры функции –
// имя открываемого файла, количество элементов,
// записанных в файле.
void change_f(char *name,int n){
 int i= 0;
 int z,y;
 int j = (n-1)*sizeof(int);
// i – номер позиции в начале файла.
// j – номер позиции в конце файла.
 FILE *f = fopen(name, "r+b");
// Всего необходимо провести n/2 обменов.
 for(int k=0;k<n/2;k++)
 {
// Прочитать число с позиции i в переменную z.
   fseek(f,i,SEEK_SET);
   fread(&z,sizeof(z),1,f);
// Прочитать число с позиции j в переменную y.
   fseek(f,j,SEEK_SET);
   fread(&y,sizeof(y),1,f);
// Записать число z на позицию j.
   fseek(f,i,SEEK_SET);
   fwrite(&y,sizeof(y),1,f);
 // Записать число y на позицию i.

  fseek(f,j,SEEK_SET);
   fwrite(&z,sizeof(z),1,f);
// Увеличить номер позиции i, уменьшить номер позиции j.
   i+=sizeof(int);
   j-=sizeof(int);
 }
}
void main()
 {
 clrscr();
// Прочитать файл  My_int.fil, количество записей в файле
// сохранить в переменную k.
int k = RP_f("My_int.fil");
//  «Перевернуть» файл
 change_f("My_int.fil",k);
// Напечатать измененный файл
 k = RP_f("My_int.fil");
 getch();
```

}

Порядок выполнения работы

1. Получить индивидуальное задание
2. Составить и записать алгоритм решения задачи
3. Составить программу, реализующую алгоритм.
4. Выполнить компиляцию проекта
5. Написать отчет о проделанной работе.
6. Защитить работу

Задание оценивается в 7 баллов: 2 балла алгоритм,
1 балл код программы.
2 балла отчет.
2 балла защита.

Содержание отчета:

1.Текст задания
2. Алгоритм
3. Тестирование программы

Лабораторная работа №8. *Списки*

Цель работы – научиться работать с динамическими структурами данных.

Динамические структуры данных

Часто в серьезных программах надо использовать данные, размер и структура которых должны меняться в процессе работы. Динамические массивы здесь не выручают, поскольку заранее нельзя сказать, сколько памяти надо выделить – это выясняется только в процессе работы. Например, надо проанализировать текст и определить, какие слова и в каком количество в нем встречаются, причем эти слова нужно расставить по алфавиту.

В таких случаях применяют данные особой структуры, которые представляют собой отдельные элементы, связанные с помощью ссылок.

Каждый элемент (узел) состоит из двух областей памяти: поля данных и ссылок (рис. 1). Ссылки – это адреса других узлов этого же типа, с которыми данный элемент логически связан. В языке Си для организации ссылок используются переменные-указатели. При добавлении нового узла в такую структуру выделяется новый блок памяти и (с помощью ссылок) устанавливаются связи этого элемента с уже существующими. Для обозначения конечного элемента в цепи используются нулевые ссылки (NULL).

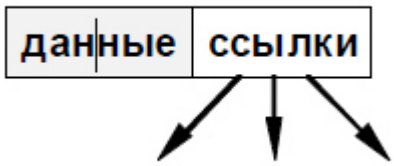


Рис. 8.1. Структура узла списка

Линейный список

В простейшем случае каждый узел содержит всего одну ссылку. Для определенности будем считать, что решается задача частотного анализа текста – определения всех слов, встречающихся в тексте и их количества. В этом случае область данных элемента включает строку (длиной не более 40 символов) и целое число.

Каждый элемент содержит также ссылку на следующий за ним элемент. У последнего в списке элемента поле ссылки содержит NULL. Чтобы не потерять список, мы должны где-то (в переменной) хранить адрес его первого узла – он называется «головой» списка (рис. 2).

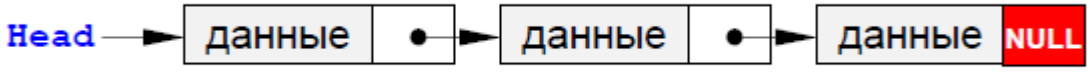


Рис. 8.2. Структура линейного списка

В программе надо объявить два новых типа данных – узел списка Node и указатель на него PNode. Узел представляет собой структуру, которая содержит три поля - строку, целое число и указатель на такой же узел. Правилами языка Си допускается объявление:

```
struct Node {
char word[40]; // область данных
int count;
Node *next; // ссылка на следующий узел
};
typedef Node *PNode; // тип данных: указатель на узел
```

В дальнейшем мы будем считать, что указатель ***Head*** указывает на начало списка, то есть, объявлен в виде

*PNode Head = NULL***;**

В начале работы в списке нет ни одного элемента, поэтому в указатель ***Head*** записывается нулевой адрес *NULL*.

Создание элемента списка

Для того, чтобы добавить узел к списку, необходимо создать его, то есть выделить память под узел и запомнить адрес выделенного блока. Будем считать, что надо добавить к списку узел, соответствующий новому слову, которое записано в переменной NewWord. Составим функцию, которая создает новый узел в памяти и возвращает его адрес.

Обратите внимание, что при записи данных в узел используется обращение к полям структуры через указатель.

```
PNode CreateNode ( char NewWord[] )
{
PNode NewNode = new Node; // указатель на новый узел
strcpy(NewNode->word, NewWord); // записать слово
NewNode->count = 1; // счетчик слов = 1
NewNode->next = NULL; // следующего узла нет
return NewNode; // результат функции – адрес узла
}
```

После этого узел надо добавить к списку (в начало, в конец или в середину).

Добавление узла

Добавление узла в начало списка

При добавлении нового узла NewNode в начало списка надо установить ссылку узла NewNode на голову существующего списка (рис. 3.1) и установить голову списка на новый узел (рис. 3.2).

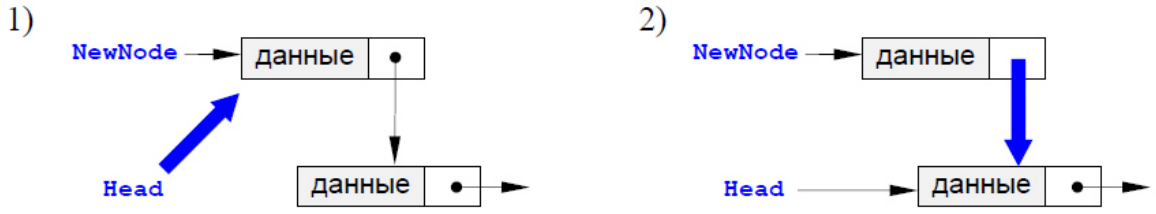


Рис.8.3. Добавление узла в начало списка

По такой схеме работает процедура *AddFirst*. Предполагается, что адрес начала списка хранится в *Head*. Важно, что здесь и далее адрес начала списка передается по ссылке, так как при добавлении нового узла он изменяется внутри процедуры.

```
void AddFirst (PNode &Head, PNode NewNode)
{
NewNode->next = Head;
Head = NewNode;
}
```

Добавление узла после заданного

Дан адрес *NewNode* нового узла и адрес p одного из существующих узлов в списке. Требуется вставить в список новый узел после узла с адресом *p*. Эта операция выполняется в два этапа:

1) установить ссылку нового узла на узел, следующий за данным;

2) установить ссылку данного узла p на NewNode (рис. 4).

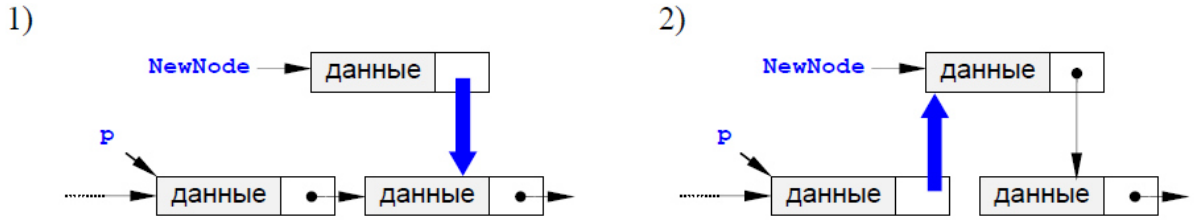


Рис. 8.4. Добавление узла после заданного

Последовательность операций менять нельзя, потому что если сначала поменять ссылку у узла *p*, будет потерян адрес следующего узла.

```
void AddAfter (PNode p, PNode NewNode)
{
NewNode->next = p->next;
p->next = NewNode;
}
```

Добавление узла перед заданным

Эта схема добавления самая сложная. Проблема заключается в том, что в простейшем линейном списке (он называется односвязным, потому что связи направлены только в одну сторону) для того, чтобы получить адрес предыдущего узла, нужно пройти весь список сначала.

Задача сведется либо к вставке узла в начало списка (если заданный узел – первый), либо к вставке после заданного узла.

```
void AddBefore(PNode &Head, PNode p, PNode NewNode)
{
PNode q = Head;
if (Head == p) {
AddFirst(Head, NewNode); // вставка перед первым узлом
return;
}
while (q && q->next!=p) // ищем узел, за которым следует p
q = q->next;
if ( q ) // если нашли такой узел,
AddAfter(q, NewNode); // добавить новый после него
}
```

Существует еще один интересный прием: если надо вставить новый узел *NewNode* до заданного узла *p*, вставляют узел после этого узла, а потом выполняется обмен данными между узлами *NewNode* и *p*. Таким образом, по адресу *p* в самом деле будет расположен узел с новыми данными, а по адресу *NewNode* – с теми данными, которые были в узле *p* Этот прием не сработает, если адрес нового узла NewNode запоминается где-то в программе и потом используется, поскольку по этому адресу будут находиться другие данные.

Добавление узла в конец списка

Для решения задачи надо сначала найти последний узел, у которого ссылка равна NULL, а затем воспользоваться процедурой вставки после заданного узла. Отдельно надо обработать случай, когда список пуст.

```
void AddLast(PNode &Head, PNode NewNode)
{
PNode q = Head;
if (Head == NULL) { // если список пуст,
AddFirst(Head, NewNode); // вставляем первый элемент
return;
```

```
}
while (q->next) q = q->next; // ищем последний элемент
AddAfter(q, NewNode);
}
```

Проход по списку

Для того, чтобы пройти весь список и сделать что-либо с каждым его элементом, надо начать с головы и, используя указатель *next*, продвигаться к следующему узлу.

```
PNode p = Head; // начали с головы списка
while ( p != NULL ) { // пока не дошли до конца
// делаем что-нибудь с узлом p
p = p->next; // переходим к следующему узлу
}
```

Поиск узла в списке

Часто требуется найти в списке нужный элемент (его адрес или данные). Надо учесть, что требуемого элемента может и не быть, тогда просмотр заканчивается при достижении конца списка. Такой подход приводит к следующему алгоритму:

1) начать с головы списка;

2) пока текущий элемент существует (указатель – не NULL), проверить нужное условие и перейти к следующему элементу;

3) закончить, когда найден требуемый элемент или все элементы списка просмотрены.

Например, следующая функция ищет в списке элемент, соответствующий заданному слову (для которого поле *word* совпадает с заданной строкой *NewWord*), и возвращает его адрес или NULL, если такого узла нет.

```
PNode Find (PNode Head, char NewWord[])
{
PNode q = Head;
while (q && strcmp(q->word, NewWord))
q = q->next;
return q;
}
```

Удаление узла

Эта процедура также связана с поиском заданного узла по всему списку, так как нам надо поменять ссылку у предыдущего узла, а перейти к нему непосредственно невозможно. Если мы нашли узел, за которым идет удаляемый узел, надо просто переставить ссылку (рис. 5).

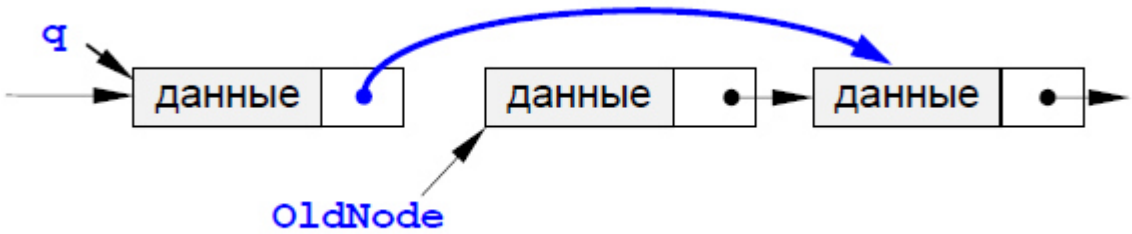


Рис. 8.5. Удаление узла

Отдельно обрабатывается случай, когда удаляется первый элемент списка. При удалении узла освобождается память, которую он занимал.

Отдельно рассматриваем случай, когда удаляется первый элемент списка. В этом случае адрес удаляемого узла совпадает с адресом головы списка *Head* и надо просто записать в *Head* адрес следующего элемента.

```
void DeleteNode(PNode &Head, PNode OldNode)
{
```

```
PNode q = Head;
if (Head == OldNode)
Head = OldNode->next; // удаляем первый элемент
else {
while (q && q->next != OldNode) // ищем элемент
q = q->next;
if ( q == NULL ) return; // если не нашли, выход
q->next = OldNode->next;
}
delete OldNode; // освобождаем память
}
```

Барьеры

Вы заметили, что для рассмотренного варианта списка требуется отдельно обрабатывать граничные случаи: добавление в начало, добавление в конец, удаление одного из крайних элементов. Можно значительно упростить приведенные выше процедуры, если установить два барьера – фиктивные первый и последний элементы. Таким образом, в списке всегда есть хотя бы два элемента-барьера, а все рабочие узлы находятся между ними.

Двусвязный список

Многие проблемы при работе с односвязным списком вызваны тем, что в них невозможно перейти к предыдущему элементу. Возникает естественная идея – хранить в памяти ссылку не только на следующий, но и на предыдущий элемент списка. Для доступа к списку используется не одна переменная-указатель, а две – ссылка на «голову» списка (*Head*) и на «хвост» - последний элемент (*Tail*) (рис. 6).

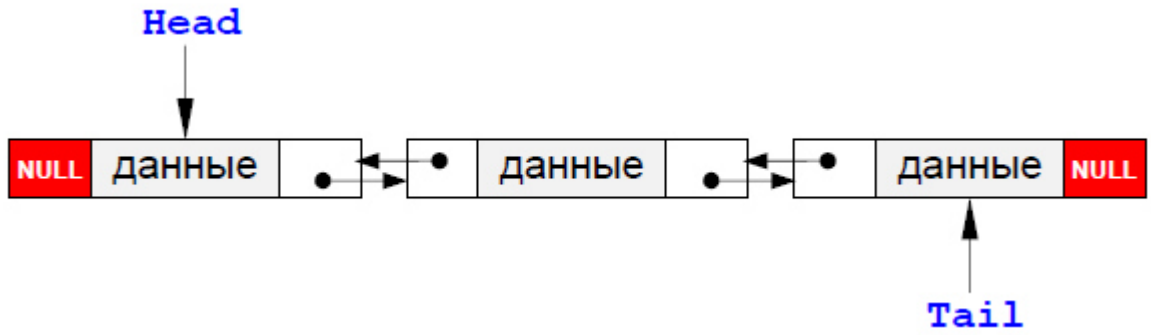


Рис. 8.6. Двунаправленный список

Каждый узел содержит (кроме полезных данных) также ссылку на следующий за ним узел (поле *next*) и предыдущий (поле *prev*). Поле *next* у последнего элемента и поле *prev* у первого содержат NULL. Узел объявляется так:

```
struct Node {
char word[40]; // область данных
int count;
Node *next, *prev; // ссылки на соседние узлы
};
typedef Node *PNode; // тип данных «указатель на узел»
```

В дальнейшем мы будем считать, что указатель *Head* указывает на начало списка, а указатель *Tail* – на конец списка:

```
PNode Head = NULL, Tail = NULL;
```

Для пустого списка оба указателя равны *NULL*.

Операции с двусвязным списком

Добавление узла в начало списка

При добавлении нового узла *NewNode* в начало списка надо

1) установить ссылку next узла *NewNode* на голову существующего списка и его ссылку prev в *NULL*;

2) установить ссылку prev бывшего первого узла (если он существовал) на *NewNode*;

3) установить голову списка на новый узел;

4) если в списке не было ни одного элемента, хвост списка также устанавливается на новый узел (рис. 7).

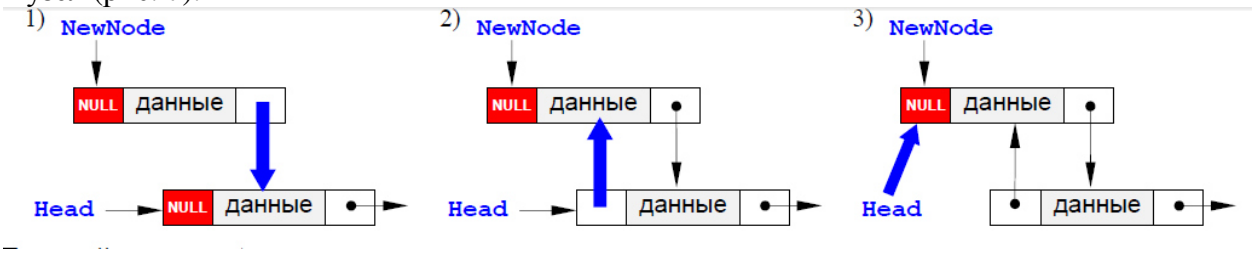


Рис.8.7. Добавление узла в начало списка

По такой схеме работает следующая процедура:

```
void AddFirst(PNode &Head, PNode &Tail, PNode NewNode)
{
NewNode->next = Head;
NewNode->prev = NULL;
if ( Head ) Head->prev = NewNode;
Head = NewNode;
if ( ! Tail ) Tail = Head; // этот элемент – первый
```

Добавление узла в конец списка

Благодаря симметрии добавление нового узла *NewNode* в конец списка проходит совершенно аналогично, в процедуре надо везде заменить *Head* на *Tail* и наоборот, а также поменять *prev* и *next*.

Добавление узла после заданного

Дан адрес *NewNode* нового узла и адрес *p* одного из существующих узлов в списке. Требуется вставить в список новый узел после *p*. Если узел *p* является последним, то операция сводится к добавлению в конец списка (см. выше). Если узел *p* – не последний, то операция вставки выполняется в два этапа:

1) установить ссылки нового узла на следующий за данным (*next*) и предшествующий ему (*prev*);

2) установить ссылки соседних узлов так, чтобы включить *NewNode* в список (рис. 8).

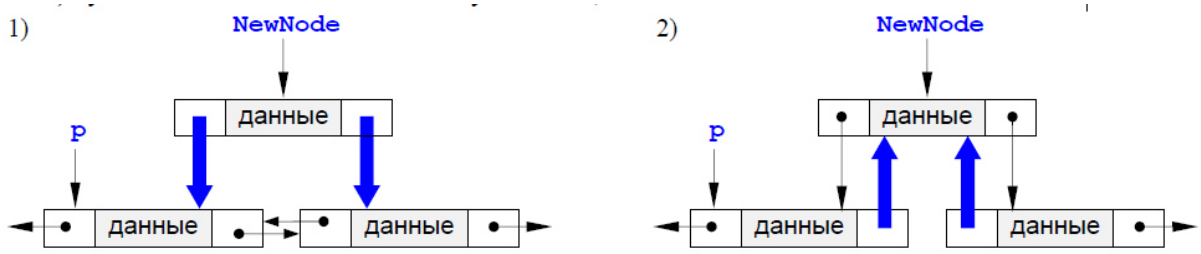


Рис. 8.8. Добавление элемента после заданного

Такой метод реализует приведенная ниже процедура (она учитывает также возможность вставки элемента в конец списка, именно для этого в параметрах передаются ссылки на голову и хвост списка):

```
void AddAfter (PNode &Head, PNode &Tail,
PNode p, PNode NewNode)
{
```

```
if ( ! p->next )
AddLast (Head, Tail, NewNode); // вставка в конец списка
else {
NewNode->next = p->next; // меняем ссылки нового узла
NewNode->prev = p;
p->next->prev = NewNode; // меняем ссылки соседних узлов
p->next = NewNode;
}
}
```

Добавление узла перед заданным выполняется аналогично.

Поиск узла в списке

Проход по двусвязному списку может выполняться в двух направлениях – от головы к хвосту (как для односвязного) или от хвоста к голове.

Удаление узла

Эта процедура также требует ссылки на голову и хвост списка, поскольку они могут измениться при удалении крайнего элемента списка. На первом этапе устанавливаются ссылки соседних узлов (если они есть) так, как если бы удаляемого узла не было бы. Затем узел удаляется и память, которую он занимает, освобождается. Эти этапы показаны на рисунке 9. Отдельно проверяется, не является ли удаляемый узел первым или последним узлом списка.

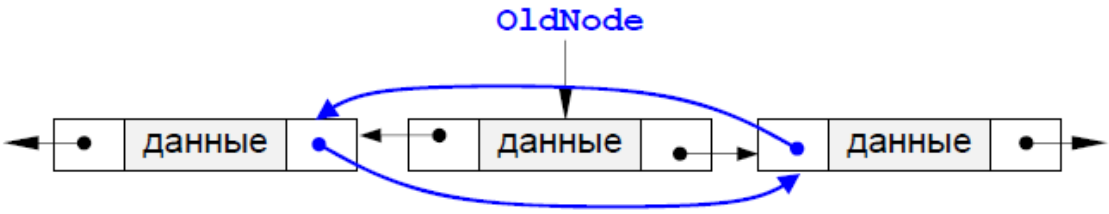


Рис. 8.9. Удаление узла.

```
void Delete(PNode &Head, PNode &Tail, PNode OldNode)
{
if (Head == OldNode) {
Head = OldNode->next; // удаляем первый элемент
if ( Head )
Head->prev = NULL;
else Tail = NULL; // удалили единственный элемент
}
else {
OldNode->prev->next = OldNode->next;
if ( OldNode->next )
OldNode->next->prev = OldNode->prev;
else Tail = NULL; // удалили последний элемент
}
delete OldNode;
}
```

Циклические списки

Иногда список (односвязный или двусвязный) замыкают в кольцо, то есть указатель *next* последнего элемента указывает на первый элемент, и (для двусвязных списков) указатель *prev* первого элемента указывает на последний. В таких списках понятие «хвоста» списка не имеет смысла, для работы с ним надо использовать указатель на «голову», причем «головой» можно считать любой элемент.

Порядок выполнения работы

1. Получить индивидуальное задание
2. Составить и записать алгоритм решения задачи
3. Составить программу, реализующую алгоритм.
4. Выполнить компиляцию проекта
5. Написать отчет о проделанной работе.
6. Защитить работу

Задание оценивается в 7 баллов: 2 балла алгоритм,
1 балл код программы.
2 балла отчет.
2 балла защита.

Содержание отчета:

1.Текст задания
2. Алгоритм
3. Тестирование программы

Лабораторная работа № 9. Функции. Управление выводом в консоли.

Цель работы – научиться применять функции WIN API для управления выводом в консольное окно.

Функции API для управления выводом в консоли.

Анализ нажатой клавиши.
Функция GetAsyncKeyState()
Описание: *int GetAsyncKeyState(int Key);*

Определяет состояние виртуальной клавиши.
Параметры:
Key: Код виртуальной клавиши.
Возвращаемое значение:
Если установлен старший байт, клавиша Key находится в нажатом положении, если младший - то клавиша Key была нажата после предыдущего вызова функции.

Функция keybd_event()
Функция keybd_event синтезирует нажатие клавиши. Вызывает функцию keybd_event программа обработки прерываний драйвера клавиатуры.
Описание:

```
void keybd_event(
   BYTE bVk,
   BYTE bScan,
   DWORD dwFlags,
   PTR dwExtraInfo
);
```

Параметры
bVk Определяет код виртуальной клавиши. Код должен быть значением в диапазоне от 1 до 254.

bScan не используется.
dwFlags Определяет различные виды операций функции. Этот параметр может состоять из одного или нескольких следующих значений:
KEYEVENTF_EXTENDEDKEY - Если он установлен, скэн-коду предшествует префиксный байт, имеющий значение 0xE0 (224).
KEYEVENTF_KEYUP Если он установлен, клавиша была отпущена. Если не установлен, клавиша была нажата.
dwExtraInfo Определяет дополнительное значение, связанное с нажатием клавиши.
Возвращаемое значение
нет возвращаемых значений.

Коды клавиш описаны в файле ***Коды клавиш.doc***
Пример использования функций в папке ***PressButton***.

Управление курсором.

Функция GetStdHandle
Описание: HANDLE GetStdHandle(HANDLE hCon); - получить дескриптор консоли
Параметры
Дескриптор hCon

Тип COORD – структура для описания позиции курсора, содержащая два поля **X** и **Y**

Функция SetConsoleCursorPosition()

Описание: SetConsoleCursorPosition(HANDLE hCon, COORD cPos); - установить курсор на позицию cPos.

Пример использования функций - папка ENUM

Изменение цвета текста.

В файле windows.h определены три константы – коды основных цветов

FOREGROUND_RED
FOREGROUND_GREEN
FOREGROUND_BLUE

и константа, изменяющая интенсивность выводимых символов -
FOREGROUND_INTENSITY

Все остальные цвета формируются комбинациями 4-х значений.

Комбинации констант для задания цветов – файл Цвета.doc

Изменение цвета фона.

Принцип определения цвета фона аналогичен принципу определения цвета текста, для комбинаций используются константы

BACKGROUND_RED

BACKGROUND_GREEN
BACKGROUND_BLUE
BACKGROUND_INTENSITY

По умолчанию – цвет фона – черный, цвет символов – белый.

Для изменения цвета используются следующие типы и функции:

Тип
typedef struct _CONSOLE_SCREEN_BUFFER_INFO {
COORD dwSize; - размер окна консоли
COORD dwCursorPosition; - текущие координаты курсора
WORD wAttributes; - атрибуты цвета
SMALL_RECT srWindow; - местоположение консоли
COORD dwMaximumWindowSize; - максимальный размер окна
} **CONSOLE_SCREEN_BUFFER_INFO** ;

Функция:

BOOL **SetConsoleTextAttribute**(

HANDLE hConsoleOutput, // дескриптор экранного буфера
WORD wAttributes // цвет текста и фона

);
Функция устанавливает значение цвета и фона, заданные в wAttributes.

Например:
SetConsoleTextAttribute(hCon, FOREGROUND_RED| FOREGROUND_INTENSITY) – Установлены текстовые атрибуты – цвет символов ярко-красный, цвет фона черный (по умолчанию).

Пример использования папка Меню1.

Порядок выполнения работы

1. Получить индивидуальное задание
2. Составить и записать алгоритм решения задачи
3. Составить программу, реализующую алгоритм.
4. Выполнить компиляцию проекта
5. Написать отчет о проделанной работе.
6. Защитить работу

Задание оценивается в 5 баллов: 1 балл алгоритм,
1 балл код программы.
1 балл отчет.
2 балла защита.

Содержание отчета:

1.Текст задания

2. Алгоритм
3. Тестирование программы

Лабораторная работа № 10. «Поразрядная сортировка»

Цель работы

Ознакомиться и реализовать алгоритмы поразрядной *LSD* и *MSD* сортировок.

Поразрядная сортировка

На практике сортировка применяется в основном к каким-либо структурам данных и выполняется по определенному ключу. Природа ключей может быть очень сложной. Совсем не обязательно, что на каждом шаге обрабатывается весь ключ.

Рассмотрим пример: известен автор – по трем первым буквам выбирается ящик каталога и в нем ведется поиск. Для того, чтобы сортировки были такими же эффективными будем рассматривать структуру ключей. Т.е. рассмотрим ключи как последовательности –

- Строки - последовательности символов
- Двоичные числа – последовательности битов
- Десятичные числа – последовательности десятичных разрядов.

Каждый элемент такой последовательности имеет строго определенный размер. Сортировки, основанные на обработке за раз одного такого элемента называются поразрядными *(radix).*

Например, сортировка абонентов библиотеки – библиотекарь выставляет карточку клиента в отделение, на котором обозначена одна буква фамилии (всего отделений может быть 29). Если карточек много, то внутри отделения возможна сортировка по второй букве и т.д.. Это пример поразрядной сортировки с основанием 29.

Основа поразрядной сортировки – извлечь *i*- тый объект последовательности.

Существуют два базовых подхода: первый анализирует объекты слева направо (первыми обрабатываются наиболее значащие цифры). Такая сортировка называется *MSD* (*most significant digit*)- сортировкой.

Второй подход анализирует цифры справа налево (первыми обрабатываются меньшие разряды) – *LSD* (*last significant digit*).

Средства Си для реализации поразрядных сортировок

Для выделения *i* – того разряда десятичного числа *x* можно воспользоваться формулой $(x/R^i) \bmod R$. Например, извлечем разряд сотен числа 2875: **(2875/100) mod 10 →28 mod 10 ==8**

Для выделения символа строки используется обращение по индексу. Например, *char *x* – строка, *x[i]* – *i*-тый символ строки

При выделении *i*- того двоичного разряда можно использовать следующий алгоритм:

1. Выполнить сдвиг вправо на *i.*
2. Наложить на исходное число маску 1 (выполнить логическое умножение).
3. Полученное число вернуть в качестве результата.

Функция, выделяющая разряд может быть записана следующим образом:

```
// x - исходное число, d – номер разряда
int digit(int x, int d)
{ int k = x>>d;
  k = k&1;
  return k;}
```

MSD - сортировка

Поразрядная сортировка

Предположим, следующие латинские буквы имеют коды:

a 000	**b 001**	**c 010**	**d 011**
e 100	**f 101**	**g 110**	**h 111**

Дана последовательность *e f d e c g h d e e*, необходимо упорядочить ее. Просмотрим последовательность слева направо, и найдем первый ключ, который начинается с бита 1, далее просмотрим последовательность справа налево и найдем ключ, начинающийся с бита 0. Обменяем их местами и будем выполнять этот процесс пока индексы просмотров не пересекутся:

100	011	011
101	**101**	010
011	011	011
100	100	100
010 →	**010** →	101
110	110	110
111	111	111
011	100	100
100	100	100
100	100	100

Первый шаг сортировки закончен. Получено два подмассива: с первой единицей и с первым нулем.

Рекурсивно применим ту же самую процедуру к полученным массивам по следующему разряду:

011	
010	
011	
100	
101	Без изменений, все элементы разряда равны 1.
110	
111	
100	
100	
100	

По третьему разряду:

011	010	
010	011	
011	011	
100	100	
101	101	Сортировка первого подмассси-
110	110	ва закончена, так как рассмотрен
111	111	последний разряд
100	100	
100	100	
100	100	

Начнем сортировать по второму разряду второй подмассив:

011	010	010
010	011	011
011	011	011
100	100	100
101	101	101
110	100	100
111	**111**	100
100	100	100
100	**100**	111
100	110	110

Сортировка по третьему разряду двух полученных подмассивов:

010	010	010
011	011	011
011	011	011
100	100	100
101	100	100
100	100	100
100	100	100
100	101	101
111	**111**	**110**
110	**110**	**111**

Сортировка закончена, получен отсортированный массив:
c d d e e e ef g h.

Алгоритм сортировки в общем виде можно записать следующим образом:

```
Radix_bin(X,F,L,d)
{
        Если d<0 или F==L то выход
        I=F, J = L;
        1: пока I<J
        если байт d в x[I] == 0 то I++
        пока I<J
          если байт d в x[j] == 1 то J++
        Если I<J то меняем местами x[I] и x[J]
        возврат на 1.
        radix_bin(X,F,J,d-1)
        radix_bin(X,I,L,d-1)
  }
```

Поразрядная MSD сортировка

Если в быстрой двоичной сортировке разделение происходит на два подмассива (0 и 1), то *MSD* сортировка обобщает понятие поразрядной сортировки по произвольному основанию *R* – происходит разделение всего массива на *R* подмассивов.

Таким образом, в функции будет выполнено *R* рекурсивных вызовов.

Для больших значений *R* можно использовать следующую схему передвижения элементов к своему подмассиву. Создаются два вспомогательных массива – массив счетчиков для каждого из значений *R* и временный массив для хранения передвинутых элементов. Просматривается исходный массив первый раз и подсчитывается сколько раз встретилась буква *“а”* среди первых букв слов, буква *“б”* и т.д.. При втором проходе элементы из исходного массива записываются во временный, используя те точки разделения, которые получены при

первом проходе. Для вычисления точек разделения можно использовать следующий алгоритм:

3 4 2 7 9 → 3 7 9 16 25

Каждый элемент массива сложим с предыдущим. Далее просмотрим основной массив – если первая буква *"a"*, то ставим ее на *2 (3-1)* место и первый элемент массива разделений уменьшаем: *2 7 9 16 25* и т.д..

Основная часть работы происходит на первом же этапе разделения. Можно улучшить алгоритм *MSD* если для сортировки подмассивов маленькой размерности использовать алгоритм простой сортировки.

Поразрядная LSD сортировка

Сортировка работает только в случае устойчивого способа перестановки элементов. К устойчивым сортировкам относится сортировка вставками, поэтому можно применить его. Рассмотрим на примере:

01001	10010	10100	00000	00000	00000	0
10010	10110	00000	10000	10000	00011	3
11011	10100	10000	01001	10010	01001	9
10110	00000	01001	10010	00011	10000	16
00011	10000	10010	11011	10100	10010	18
10100	01001	10110	00011	10110	10100	20
00000	11011	11011	10100	01001	10110	22
10000	00011	00011	10110	11011	11011	27

Сортировка нерекурсивная.

На основании *R* выполняется аналогичным образом и остается такой же нерекурсивной.

Порядок выполнения

- Получите вариант задания у преподавателя.
- Составьте алгоритм поразрядной сортировки.
- Реализуйте алгоритм на языке Си.
- Проанализируйте полученные результаты.

Лабораторная работа № 11. Организация поиска элементов по заданному ключу

Цель работы

Ознакомиться с основными алгоритмами поиска элементов в массиве. Реализовать на языке Си различные методы поиска.

Прямой поиск

Задача поиска элемента в какой-либо структуре данных сводится к просмотру элементов структуры и последующему возвращению места нахождения искомого элемента. Для данных, хранимых в массивах, это будет индекс. Для динамических структур данных это будет адрес (указатель). Для различных прикладных задач можно модифицировать алгоритм поиска:

- найти количество заданных элементов;
- если заданных элементов несколько, то найти все элементы;

- если заданных элементов несколько, то найти первое вхождение заданного элемента и т.д..

Рассмотрим все алгоритмы для следующей формулировки задачи поиска: найти первое вхождение заданного элемента.

Самый простой и неэффективный метод поиска – прямой поиск. Его алгоритм может быть описан следующим образом:

Поиск(массив *X,* элемент *a*)
{n – размер массива *X;*
ЦИКЛ($i = \overline{0;n}$)
ЕСЛИ (*X[i]==a*) вернуть *i;*
вернуть -1;
}

Функция возвращает -1, если поиск прошел неуспешно. В случае успеха функция возвращает индекс заданного элемента.

Очевидно, что время поиска прямо пропорционально размеру просматриваемого массива.

Для улучшения прямого поиска можно воспользоваться следующими стратегиями:

- пусть в системе высока вероятность того, что ранее найденный элемент будет и далее участвовать в поиске, тогда можно переносить найденный элемент в начало массива; тогда следующий поиск пройдет быстрее;
- пусть в системе высока вероятность того, что ранее найденный элемент уже не будет использоваться в поиске, в этом случае перенесем элемент в конец массива.

Бинарный поиск

Если данные массива упорядочены, то можно сократить время поиска, реализуя следующий алгоритм:

1. *X* – массив, *n* – размерность массива, *a* – значение для поиска.
2. *F = 0, L = n-1*
3. *M = (F+L)/2 //* найдем середину массива
4. ЕСЛИ (*a==X[M]*) ТО вернуть *M //* элемент найден
5. ЕСЛИ (*a<X[M]*) ТО *L = M-1 //* далее ищем в левой половине
6. ЕСЛИ (*a]>X[M]*) ТО *F = M+1 //* далее ищем в правой половине
7. ЕСЛИ (*F<L*) ТО вернуться на шаг 3.
8. Вернуть -1 // поиск прошел неуспешно.

Время бинарного поиска прямо пропорционально $\log_2 n$. Т.е. если рассматривается массив из 32 элементов то в самом худшем случае будет выполнено 5 сравнений, в то время как при прямом поиске будет выполнено 32 сравнения.

Интерполяционный поиск

Этот поиск является улучшенной версией бинарного поиска для числовых отсортированных массивов. Предположим, что данные в массиве равномерно распределены на промежутке от 0 и до *n.* Очевидно, что чем больше значение, тем ближе оно находится к концу массива. Интерполяционный поиск позволяет определить размер массива, в котором будет производиться поиск в зависимости от значения искомого элемента. То есть, если ищется маленькое значение, то точка разделения будет находиться ближе к началу массива. При поиске больших значений точка разделения сдвинется к концу массива.

Алгоритм интерполяционного поиска выглядит следующим образом:

1. *X* – массив, *n* – размерность массива, *a* – значение для поиска.
2. *F = 0, L = n-1*
3. *M = F+(a – x[F])/(X[L]-X[F])(L-F)//* найдем точку разделения
4. ЕСЛИ (*a==X[M]*) ТО вернуть *M //* элемент найден
5. ЕСЛИ (*a<X[M]*) ТО *L = M-1 //* далее ищем в левой половине
6. ЕСЛИ (*a>X[M]*) ТО *F = M+1 //* далее ищем в правой половине
7. ЕСЛИ (*F<L*) ТО вернуться на шаг 3.
8. Вернуть -1 // поиск прошел неуспешно.

Порядок выполнения работы

- Получите вариант задания у преподавателя.
- Составьте алгоритм поиска элемента по заданному значению.
- Реализуйте алгоритм на языке Си таким образом, программа искала индексы всех имеющихся в массиве заданных элементов.
- Проанализируйте полученные результаты.

Лабораторная работа № 12. Организация поиска подстроки в строке

Цель работы

Ознакомиться и реализовать алгоритмы поиска подстроки в строке.

Последовательный (прямой) поиск

Алгоритм прямого поиска подстроки подобен алгоритму прямого поиска элемента с заданным значением в массиве.

Пусть *T* – текст, *O* – подстрока поиска. Длина текста равна *m,* длина подстроки *n.*

1. *i = 0*; // счетчик символов текста
2. ПОКА(*i<m*)

k = 0; // счетчик совпадений
j=0; // счетчик символов подстроки
i1 = i;
ПОКА(*O[j]==T[i1]&& k<n && i1<m*)
2.4.1. *k++;*
2.4.2. *j++,i1++;*
2.5. Конец цикла
2.6. ЕСЛИ (*k==n*) ТО вернуть *i;//* успех при поиске
2.7. *i++;*

3. Конец цикла
4. Вернуть -1. // неуспех при поиске

В худшем случае, время поиска в этом алгоритме прямо пропорционально *n*(m-1).*

Алгоритм поиска Кнута

Алгоритм, предложенный Кнутом для осуществления поиска, основан на анализе символов подстроки. В зависимости от вида подстроки и количества имеющихся совпадений алгоритм позволяет выполнить сдвиг по тексту более чем на один символ.

Вся основная работа проводится в начале алгоритма. Если в поиске участвует подстрока длины *n,* то может быть 0, 1, 2, 3, …, *n-1* совпадений с текстом (как только произойдет *n*

совпадений - подстрока найдена). Обозначим за j количество совпадений. Для дальнейшей работы рассчитаем элементы массива d_j по следующему правилу:

d_j = длина максимальной подстроки до символа с номером *j*, полностью совпадающей с началом образа.

Значения d_0 и d_1 постоянны и равны соответственно -1 и 0.

Сам сдвиг рассчитывается по формуле: $shift_j = j - d_j$

Рассмотрим на примере подстроки *AABCDAABD*.

$d_0 = -1$	
$d_1 = 0$	
$d_2 = 1$	*A**A**BCDAABD*
$d_3 = 1$	*A**A**BCDAABD*
$d_4 = 1$	*A**A**BCDAABD*
$d_5 = 1$	*A**A**BCDAABD*
$d_6 = 1$	*A**A**BCD**A**ABD*
$d_7 = 2$	*AABCD**AA**BD*
$d_8 = 3$	*AABCD**AAB**D*

Предположим, что необходимо найти эту подстроку в тексте

A	A	B	D	A	A	B	C	A	A	B	C	D	A	A	B	D
A	**A**	**B**	C	D	A	A	B	D	*j=3, shift = 2*							
		A	A	B	C	D	A	A	B	D	*j=0, shift = 1*					
			A	A	B	C	D	A	A	B	D	*j=0, shift = 1*				
				A	**A**	**B**	**C**	D	A	A	B	D	*j=4, shift = 3*			
j=9, поиск окончен								**A**	**A**	**B**	**C**	**D**	**A**	**A**	**B**	**D**

Поиск Боуера-Мура

Этот поиск начинает сравнивать подстроку со строкой, начиная с последнего символа подстроки. Пусть сравнение начинается с позиции *i -1*. Если нет полного совпадения, то сдвиг можно произвести на величину $d_{sim_{i-1}}$, где sim_{i-1} - символ строки в позиции *i-1*. Сама величина сдвига определяется следующим образом:

- *d* равно длине образа, если символ не принадлежит подстроке;
- *d* равно расстоянию от самого правого в подстроке вхождения символа до ее конца.

Рассмотрим на примере подстроки «*образ*», подстрока ищется в тексте «*отказ приказ город образ пример*». Аналогично поиску Кнута таблица сдвигов рассчитывается еще до работы алгоритма и для рассматриваемого примера равна:

$d_з = 5$ $d_а = 1$ $d_р = 2$ $d_б = 3$ $d_о = 4$

Все остальные символы алфавита имеют *d* равное 5 (длине образа).

о	т	к	а	**з**		п	р	и	**к**	а	з		г	**о**	р	о	д	…
о	б	р	а	з	сдвиг на 5 позиций													
					о	б	р	а	з	сдвиг на 5 позиций								
										о	б	р	а	з	сдвиг на 4 поз.			
														о	б	р	а	з

г	о	р	о	д		о	б	р	а	з		п	р	и	м	е	р		
	о	б	р	а	з	сдвиг на 5 позиций													
						о	**б**	**р**	**а**	**з**	подстрока найдена								

Порядок выполнения работы

- Получите вариант задания у преподавателя.
- Составьте алгоритм поиска подстроки по заданному значению.
- Реализуйте алгоритм на языке Си.
- Сравните алгоритм прямого поиска с алгоритмами Кнута и Боуера-Мура. Проанализируйте полученные результаты.

Лабораторная работа № 13 -14. BST-деревья и основные операции над ними

Цель работы

Ознакомиться и реализовать алгоритмы для работы с деревьями бинарного поиска.

Деревья бинарного поиска

Для решения проблемы высоких затрат на вставку элемента рассмотрим способ хранения информации в древовидной структуре.

Дадим основные определения для дерева бинарного поиска (*binary search tree (BST)*):

– каждый узел указывается только одним узлом, который называется родительским;
– у каждого узла существует две связи – левая и правая;
– каждая связь может указывать на внешние узлы (у которых нет связей);
– узлы с двумя связями называются внутренними узлами;
– каждый внутренний узел имеет элемент со значением ключа.

BST -дерево – это бинарное дерево, с каждым из внутренних узлов которого связан ключ. При этом ключ в любом узле больше или равен ключам во всех узлах левого поддерева этого узла и меньше ключей во всех узлах правого поддерева этого узла.

Для программной реализации элемента дерева можно предложить следующую структуру:

```
struct tree
{int info; // поле для хранения ключа
 tree *l; // ссылка на левое поддерево
 tree *r; // ссылка на правое поддерево
}
```

На рис. 13.1. показан пример *BST*-дерева.

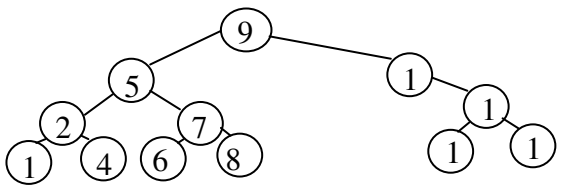


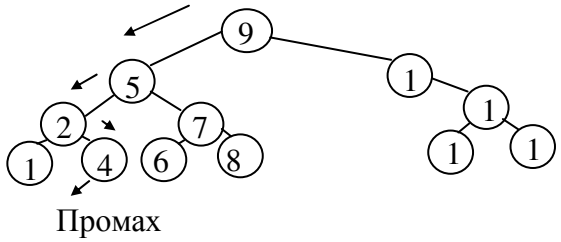
Рис. 13.1. *BST*- дерево

Поиск элемента в BST-дерево

Рекурсивный **алгоритм поиска** в *BST*-дереве выглядит следующим образом: если дерево пусто, то элемент не содержится в данном дереве (промах при поиске), если значение ключа в корне дерева равно значению искомого ключа, то имеет место успех при поиске. В другом случае происходит рекурсивный спуск в соответствующем поддереве.

Поиск (*R, Key*) // *R* – текущий узел, *key* – значение для поиска
{
Если *R* – пусто, то элемент не найден
Если *R.info == Key* То вернуть значение *R;*
Если (*R.info* > *Key*) То вернуть Поиск(*R.l, Key*)
Иначе вернуть Поиск (*R.r, Key*)
}

Первый раз функция поиска вызывается для корневого узла всего дерева.
На рис. 8.2, 8.3. показаны процессы поиска, заканчивающиеся промахом и успехом.



Промах

Рис. 13.2. Поиск в *BST*- дереве элемента с ключом 3

Для рассмотренного примера можно записать следующую цепочку рекурсивных вызовов:

Поиск(9,3) → Поиск(5,3) →Поиск(2,3) → Поиск(4,3) →
Поиск (пустой узел, 3) → Промах.

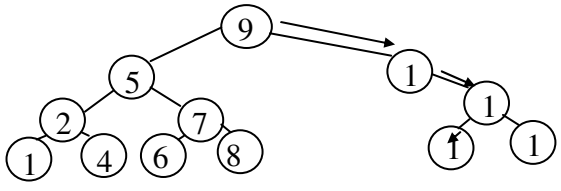


Рис. 13.3. Поиск в *BST*- дереве элемента с ключом 13

Цепочка рекурсивных вызовов для этого примера выглядит следующим образом:
Поиск(9,13) → Поиск(12,13) →Поиск(15,13) → Поиск(13,13) →
Успех.

Вставка элемента в BST - дерево

Вставка нового узла осуществляется на основе функции поиска. Как только обнаружен промах, нулевая связь заменяется на связь с вставляемым элементом.

Рекурсивная функция вставки нового элемента в *BST*-дерево может выглядеть следующим образом:

```
Добавить (R, Elem) // добавить в дерево с корнем R новый
                   //элемент с ключом Elem
  {
  Если R-  пустой
    То связать вставляемый узел с найденным внешним узлом
  Если Elem < R.info то Добавить (R.l, Elem)
    Иначе Добавить (R.r, Elem)
  }
```

На рис. 8.4. изображена вставка элемента с ключом 14. Цепочка рекурсивных вызовов выглядит следующим образом:

Добавить(9,14) → Добавить(12,14) → Добавить(15,14) →
Добавить(13,14) → Добавить(Пустой узел, 14).

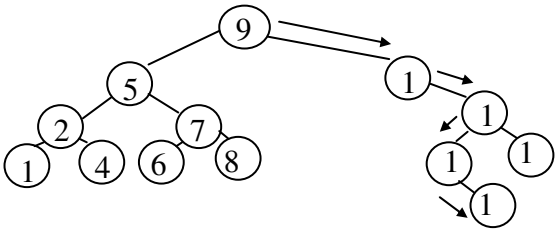


Рис. 13.4. Вставка нового элемента с ключом 14

Вставка в корень в BST –деревьях

Стандартный алгоритм вставки добавляет элементы в нижнюю часть дерева, что не всегда обязательно. Например, данные могут быть и устаревшими, а при стандартной вставке невозможно добраться до новых данных, не посетив старые данные.

Рассмотрим два случая:

- если ключ вставляемого элемента больше ключа корня, то новый элемент становится корнем, старый корень и его левое поддерево – левым поддеревом, правое поддерево старого корня – правым поддеревом. *Но в правом поддереве могут быть ключи, меньшие ключа нового элемента*.

-если ключ вставляемого элемента меньше ключа корня, то новый элемент становится корнем, старый корень и его правое поддерево – правым поддеревом, левое поддерево старого корня – левым поддеревом. *Но в левом поддереве могут быть ключи, большие ключа нового элемента*.

Перенос элементов из одного поддерева в другое относительно нового корня может быть сложной задачей. Существует способ, который сохраняет порядок ключей.

Ротация – фундаментальное преобразование деревьев, позволяет менять местами роль корня и одного из его дочерних узлов, сохраняя порядок ключей *BST* – деревьев.

Ротация вправо – участвуют корень и левый дочерний узел.

Правая связь нового корня становится левой связью старого корня, сам старый корень становится правой связью нового корня.

Ротация влево – участвуют корень и правый дочерний узел.

Левая связь нового корня становится правой связью старого корня, сам старый корень становится левой связью нового корня.

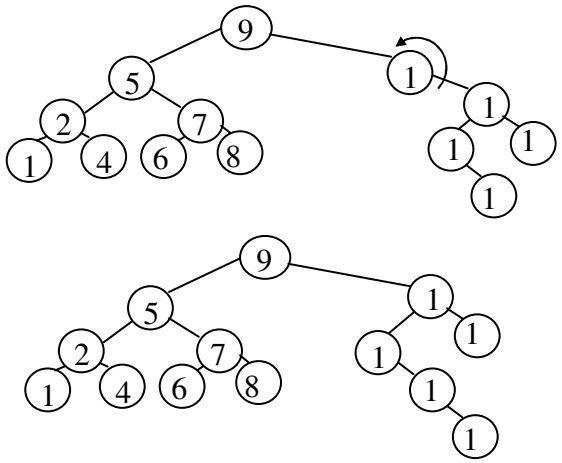


Рис. 13.5. Ротация влево в узле 12

Ротацию влево можно записать следующим образом:
h – узел, в котором происходит ротация (корень)
x = h.r; h.r = x.l; x.l = h.
После ротации корнем становится *x* (был *h*).

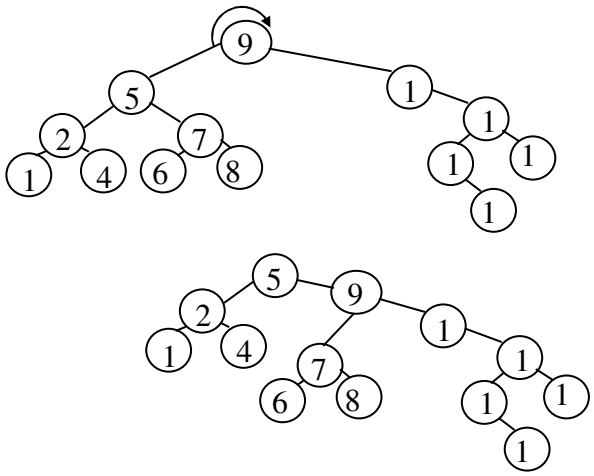


Рис.13.6. Ротация вправо в узле 9

Ротацию вправо можно записать следующим образом:
h – узел, в котором происходит ротация (корень)
X = h.l; h.l = x.r; x.r = h;

Тогда процедура вставки в корень может быть записана следующим образом:

```
Добавить  (R, Elem)
{
Если R - пустой
   То связать вставляемый узел с найденным внешним узлом
Если Elem  <  R.info То Добавить(R.l,Elem) Ротация вправо(R)
    Иначе  Добавить(R.r,Elem) Ротация влево(R)
}
```

Вывод элементов дерева

Прямой обход

Рассмотрим вывод элементов дерева на примере обхода в глубину. Функция, реализующая прямой обход выглядит следующим образом:

```
Печать (R) // R - корень дерева
    {
    Если R пустой То выход
    Вывести на печать R.info
    Печать(R.l)
    Печать(R.r)
    }
```

Рассмотрим работу функции на примере дерева, изображенного на рис. 8.6.

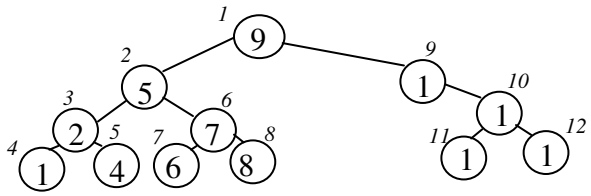


Рис. 13.7. Порядок посещения узлов дерева при прямом обходе

Симметричный обход

При симметричном обходе ключи дерева будут выведены на экран в порядке неубывания. Функция, реализующая симметричный обход выглядит следующим образом:

```
Печать (R) // R - корень дерева
    {
    Если R пустой То выход
    Печать(R.l)
    Вывести на печать R.info
    Печать(R.r)
    }
```

Рассмотрим работу функции на примере дерева, изображенного на рис. 1.7.

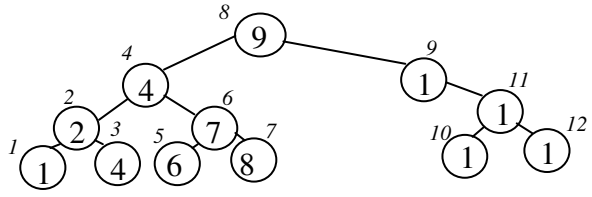


Рис. 13.8. Порядок посещения узлов дерева при симметричном обходе

Обратный обход

Еще один вариант обхода в глубину называется обратным обходом. Функция, реализующая обратный обход выглядит следующим образом:

```
Печать (R) // R -  корень дерева
    {
    Если R  пустой То выход
    Печать(R.l)
    Печать(R.r)
    Вывести на печать R.info
    }
```

Рассмотрим работу функции на примере дерева, изображенного на рис.8.8.

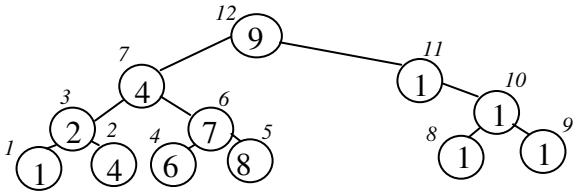


Рис. 13.9. Порядок посещения узлов дерева при обратном обходе

Порядок выполнения работы

- Получите вариант задания у преподавателя.
- Разработайте структуру данных для работы двоичным деревом
- Реализуйте на языке Си основные операции *BST*- деревом.

Лабораторная работа № 15. «Простое приложение на языке *java.* Простой апплет»

Цель работы

Ознакомиться с основами консольного и графического ввода-вывода и основными управляющими конструкциями языка *java*.

Основы языка программирования java

Структура простой программы

Язык *java* объектно-ориентированный, платформенно-независимый язык программирования. Программа на языке *java* состоит из классов, определенным образом взаимодействующих друг с другом. Один из классов обязательно должен содержать метод *main(),* который автоматически запускается интерпретатором *java.*

Простая программа на языке *java* может выглядеть следующим образом:

```
public class First{
 public static void main(String [] a){
 System.out.println("Первая программа на java");
}
```

Класс *First* содержит метод *main(),* который обязательно должен быть статическим и иметь аргументом массив строк. Вывод строки осуществляет метод *println* свойства *out* класса *System.* Класс *System* включается в пакет автоматически.

Для успешной компиляции класс *First* должен быть сохранен в файле *First.java.*

Компиляция и выполнение из командной строки

Любое приложение на *java* может быть скомпилировано из командной строки. Для этого переменная *PATH* должна быть проинициализирована строкой, указывающей месторасположение файлов *javac.exe* (компилятор) и *java.exe* (интерпретатор). Например,

SET PATH = C:\Program Files\Java\jdk1.7.0_13\bin

В различных версиях это расположение может быть различным.

Так же, необходимо задать корневой каталог для иерархии пакетов *java* с помощью переменной среды окружения *CLASSPATH:*

SET CLASSPATH = .; C:\Program Files\Java\jdk1.7.0_13\src.zip

Дополнительное значение - '.' переменной среды окружения задано для использования текущей директории в качестве рабочей.

Вызов строчного компилятора выглядит следующим образом:

javac First.java

При успешной компиляции создается виртуальный код, записывающийся в файл *First.class.*

Выполнить этот код можно с помощью интерпретатора, набрав в командной строке:

java First

Используйте *bat* – файлы для выполнения этого задания.

Компиляция и выполнение в среде eclipse

В помощь разработчикам приложений *java* существует множество интегрированных сред разработки. *Eclipse* - это расширяемая среда разработки с открытым исходным кодом.

При первом запуске загрузчика *Eclipse* перед появлением самой среды выполняется создание директории *workspase* для хранения файлов проектов.

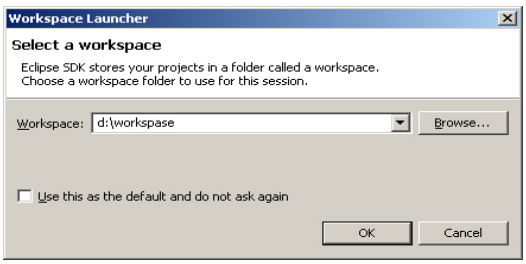


Рис.14.1. Создание директории *workspase* для хранения файлов проектов

Чтобы начать создание первого *Java*-проекта, выберите ***File->New->Project...*** В левом списке появившегося мастера выберите ***Java Project***. Затем нажимите кнопку ***Next***.

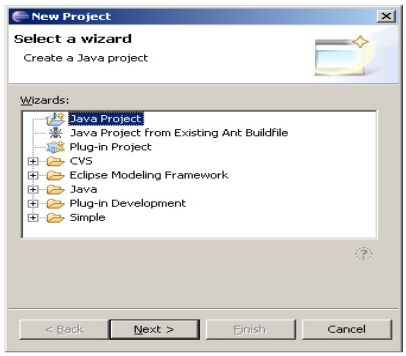


Рис. 14. 2. Первая страница мастера нового проекта

Назовем проект *MyFirstProject.*

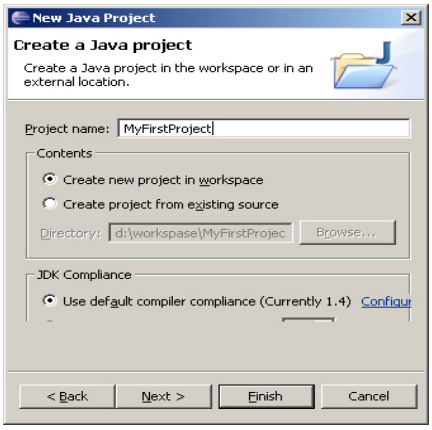
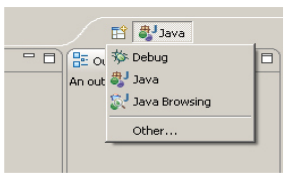


Рис. 14.3. Вторая страница мастера нового проекта

В этом же окне можно установить версию компилятора (по умолчанию 1.4) и место хранения откомпилированных файлов. После выполнения этих действий выберите кнопку *Finish,* после чего *Eclipse* создаст *java*-проект.

Новый внешний вид носит название Перспективы *Java*. Перспектива, в терминах *Eclipse*, это сохраняемый внешний вид окна, включающий любое число редакторов (editors) и представлений (views). В поставку *Eclipse* входит несколько перспектив по умолчанию (*Java, Debug, Resource* и так далее), которые можно настраивать. Перспективы управляются с помощью элементов меню ***Window*** или панели инструментов, которая обычно располагается вдоль левой границы окна *Eclipse*.



После создания проекта необходимо создать папки для хранения написанных классов. Эти папки носят название пакетов (*package*). Для создания пакета выберите *File->New->Package* или воспользуйтесь значком панели инструментов . Название пакета может быть произвольным.

После созд пакета необходимо создать класс, используя панель управления или меню *File->New->Class.* При создании класса можно установить атрибут доступа, наличие или отсутствие метода *main()* в классе. После выполнения этих действий будет создан файл <имя класса>*.java.*

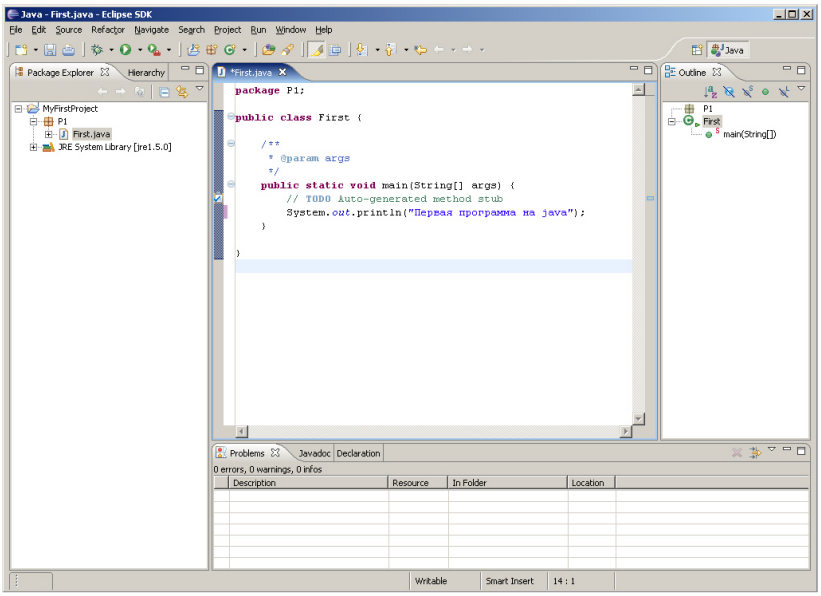


Рис. 14.4. Вид среды для перспективы *java*

Для перспективы *java* характерно следующее расположение окон:

- в левой части экрана открыт навигатор пакетов
- в средней части экрана открытые окна с текстами классов
- в правой части экрана окно, содержащее структуру активного на данный момент класса
- в нижней части экрана расположены окна сообщений об ошибках и консоль.

Для отладки и запуска программы выберите значок
или меню *Run->Run As->Java Application.*

При успешной компиляции на консоли, в нижней строке экрана появится выводимая строка.

Аргументы командной строки

Аргументы командной строки передаются программе при ее выполнении. Например, рассмотрим программу:

```
public class Two
{
 public static void main(String [] a){
// Обратите внимание на свойство length – длина массива a.
  for(int i=0;i<a.length;i++)
   System.out.println("аргумент "+i+ " " + a[i]);
}
}
```

При выполнении данной программы в командной строке

java Two programming language java

на консоль выведутся строки:

аргумент 0 *programming*

аргумент 1 *language*

аргумент 2 *java*

Типы данных

В *java* определены следующие базовые типы данных:

Тип	Размер (бит)	По умолчанию	Значения (диапазон или максимум)
boolean	8	*false*	*true, false*
byte	8	0	-128…127
char	16	‘*u*0000’	0…65535
short	16	0	-32768…32767
int	32	0	-2147483648…2147483647
long	64	0	9223720368547758*07L*
float	32	0.0*f*	3.40282347E+38
double	64	0.0	1.797693134486231570E+308

Помимо базовых типов широко используются классы-оболочки *Boolean, Character, Integer, Byte, Short, Long, Float, Double.* Эти классы находятся в библиотеке *java.lang* и являются наследниками класса *Number.* Объекты классов могут быть преобразованы к любому базовому типу методами, описанными в классе.

Строка в *java* представляет объект класса *String.* При работе со строками можно использовать перегружаемую операцию «+» - объединение строк и методы класса *String.* Строковые константы заключаются в двойные кавычки.

Управляющие конструкции

Проверка условий в *java*-программе осуществляется посредством конструкции
if (<условие>*)* <действия при истинности условного выражения>
[else <действия при ложности условия>*]*
Конструкция множественного выбора
switch (<переменная, по которой происходит выбор>*){*
case <возможное значение переменной выбора 1>*:* <действия>*; break;*
case <возможное значение переменной выбора 2>*:* <действия>*; break;*
…
default: <действия, если переменная выбора не совпала ни с одним из возможных *case*-вариантов>
}

Циклы в *java* могут быть организованы тремя способами
for(i=0;i<n;i++)
{<тело цикла>}

Переменная *i* принимает значения от 0 до *n-1(*включительно*)* с шагом 1, соответственно тело цикла выполняется *n* раз.

В следующих двух циклах тело цикла выполняется до тех пор, пока истинно условное выражение.

while (<условное выражение>*)*
{
<тело цикла>
}
do
{
<тело цикла>
}while(<условное выражение>*)*

Если тело цикла содержит одно действие, фигурные скобки могут быть опущены.

Ввод и вывод информации на консоль

Вывод информации на консоль осуществляется с помощью метода *println()* или *print()* свойства *out* класса *System.* Параметрами методов могут быть строки, данные базовых типов и объекты классов-оболочек. При сложном выводе элементы соединяются перегруженным оператором «+».

```
…
Integer I = new Integer(25);
String S = “String”;
float f = (float) 23.1;
System.out.println(S+” I = “+I+” f=”+f );
…
```

На экран выведется строка

String I = 25 f=23.1

Ввод информации в *java* выполняется с помощью объектов классов *InputStreamReader* и *BufferedReader.*

На первом шаге объявляется объект класса *InputStreamReader,* который инициализируется как стандартный объект ввода *System.in* – клавиатура.

InputStreamReader is = new InputStreamReader(System.in);

На втором шаге создается объект буферизованного ввода

BufferedReader bis = new BufferedReader(is);

Ввод информации осуществляется в строковую переменную с помощью метода *readLine().* Для перевода строкового типа в числовой тип используются методы соответствующих классов-оболочек *intValue(), floatValue()* и т.д.. Ввод информации обязательно сопровождается проверкой исключительной ситуации типа *IOException* – ошибка ввода-вывода.

Следующий фрагмент программы вводит с клавиатуры целое число *n,* размер массива строк, выделяет память под массив строк и вводит значения элементов массива строк.

```
InputStreamReader is = new InputStreamReader(System.in);
BufferedReader bis = new BufferedReader(is);
try{
      System.out.print("Enter n:");
      String nS = bis.readLine();
      n = Integer.valueOf(nS).intValue();
    } catch (IOException e){
                                System.out.print("Error"+e);
                              }
String My[]=new String[n];
for(int i=0;i<n;i++)
 {
  System.out.print("Enter a String: ");
  try{
       My[i] = bis.readLine();
    } catch (IOException e){
                                System.out.print("Error"+e);
                               }
}
```

Классы *InputStreamReader* и *BufferedReader* описаны в библиотеке *java.io.*

Для подключения библиотеки служит ключевое слово *import* <имя библиотеки>. Импорт классов из библиотеки выполняется перед описанием класса. Символ «*» означает, что из указанной библиотеки импортируются все имеющиеся в ней пакеты.

Например:
import java.io.;*

Апплеты

Апплет – небольшая программа, запускаемая *Web*-браузером. Апплеты, написанные пользователем являются наследниками (расширениями) класса *Applet* библиотеки *java.awt.** или класса *JApplet* библиотеки *javax.swing.**.

Для указания наследования в *java* используется ключевое слово *extends.*

Для запуска апплету не нужен метод *main().* Код запуска апплета помещается в метод *paint()* или *init().*

Метод *paint()* получает в качестве параметра объект класса *Graphics*. Вывод информации осуществляется методом *drawstring()* этого класса.

В классе *Graphics* определены методы:

clearRect`(int x, int y, int width, int height)` – очищает указанную прямоугольную область.

copyArea`(int x, int y, int width, int height, int dx, int dy)` – копирует указанную прямоугольную область в область с левым верхним улом *x+dx,y+dy*.

draw3DRect`(int x, int y, int width, int height, boolean raised)` – рисует 3D-прямоугольник указанного размера.

drawArc`(int x, int y, int width, int height, int startAngle, int arcAngle)` – рисует круговую или эллиптическую дугу.

drawLine`(int x1, int y1, int x2, int y2)` – рисует линию по указанным координатам.

drawOval`(int x, int y, int width, int height)` – рисует эллипс.

drawPolygon`(`*Polygon*` p)` и *drawPolygon*`(int[] xPoints, int[] yPoints, int nPoints)` – рисуют замкнутую кривую по заданным точкам.

drawPolyline`(int[] xPoints, int[] yPoints, int nPoints)` – рисует произвольную кривую линию по заданным точкам.

drawRect`(int x, int y, int width, int height)` – рисует прямоугольник.

drawString`(`*String*` str, int x, int y)` – выводит в графическом окне заданную строку.

fill3DRect`(int x, int y, int width, int height, boolean raised)` – рисует закрашенный объемный прямоугольник.

fillArc`(int x, int y, int width, int height, int startAngle, int arcAngle)` – рисует закрашенный круговой или эллиптический сектор.

fillOval`(int x, int y, int width, int height)` – рисует закрашенный эллипс.

fillPolygon`(int[] xPoints, int[] yPoints, int nPoints)` – рисует закрашенный многоугольник.

fillRect`(int x, int y, int width, int height)` – рисует закрашенный прямоугольник.

getColor`()` – возвращает текущий цвет рисования.

getFont`()` –возвращает текущий фонт.

getFontMetrics`()` – возвращает характеристики текущего фонта.

setColor`(`*Color*` c)` – устанавливает цвет рисования.

setFont`(`*Font*` font)` – устанавливает фонт.

А так же и другие методы для работы с клипами, изображениями и т.д..

Для работы с цветами в *java/awt* определен класс *Color,* полями которого являются цвета.

Для запуска апплета из среды *Eclipse* используйте меню *Run->Run As->Java Applet.*

Для того, чтобы запустить апплет с помощью браузера необходимо написать *html* – документ следующего вида:

<html><body>
<applet code = <имя апплета>>

</applet>
</body></html>

Если апплет находится в пакете, то к имени файла справа добавляется через / имя пакета. *html* документ при этом должен находиться в папке проекта. Например, в среде *Eclipse* создан проект *Progect,* в нем создан пакет *p1,* а в пакете создан апплет с именем *app.java,* скомпилированный в файл *app.class,* то имя апплета формируется следующим образом *p1/app.class,* сам *html*-документ должен находиться в папке *Progect.*

Если апплет находится не в пакете, то в имени апплета указывается только имя файла с расширением *class, html*-документ располагается в одной папке с этим файлом.

Порядок выполнения работы

- Получите вариант задания у преподавателя.
- Разработайте структуру консольного приложения и реализуйте его.
- Запустите консольное приложение из командной строки.
- Разработайте структуру апплета и реализуйте его.
- Выполните компиляцию созданных приложений из командной строки.
- Просмотрите созданый апплет в браузере.

При выполнении задания можно воспользоваться предложенными ниже примерами.

Пример java-приложения

Написать следующее консольное приложение: ввести с консоли размерность массива строк *n*. Ввести элементы массива с консоли. Найти количество элементов массива, длина которых не меньше заданной. Значение заданной длины ввести с консоли.

```
import java.io.*;
public class my1 {
public static void main(String [] args){
int n=0;
int len = 0;

InputStreamReader is = new InputStreamReader(System.in);
BufferedReader bis = new BufferedReader(is);
try{
System.out.print("Enter n:");
String nS = bis.readLine(); // чтение строки
n = Integer.valueOf(nS).intValue();// преобразование строки
в число
} catch (IOException e){
System.out.print("Error"+e);}
String My[]=new String[n]; // выделение памяти под массив
строк
for(int i=0;i<n;i++){
System.out.print("Enter a String " + i + ": ");
try{
My[i] = bis.readLine(); // чтение i-той строки с клавиатуры
} catch (IOException e){
System.out.print("Error"+e);
}}
for(int i=0;i<n;i++)
{System.out.println(My[i]); // печать полученного массива
строк
```

```
}
try{
System.out.print("Enter length:");
String nS = bis.readLine(); // чтение заданной длины
len = Integer.valueOf(nS).intValue(); // преобразование строки в число
} catch (IOException e){
System.out.print("Error"+e);}
int Sum = 0;
for (int i=0;i<n;i++){
if (My[i].length()>len) Sum++; // поиск количества слов, длина которых
// больше заданной
}
// вывод результата
System.out.println(Sum + " string`s have length more then " + len);}}
```

Пример апплета

Написать апплет с выводом в окно элементов массива. Размерность массива задается константой, элементы массива случайным образом. Выделите цветом не повторяющиеся элементы.

```
import java.awt.*;
import javax.swing.*;
public class App1 extends JApplet{
int mass[];
public void init(){
this.setBackground(Color.CYAN); // цвет фона
this.setForeground(Color.YELLOW);// цвет символов
mass = new int [10];
for(int i=0;i<mass.length;i++){
mass[i] = (int)(Math.random()*20);          }}
public void paint(Graphics G){
// вспомогательный массив меток
boolean metka []= new boolean [mass.length];
// обнулим все метки
for (int i=0;i<mass.length;i++)
 metka[i] = false;
for(int i=0;i<mass.length-1;i++)
for(int j=i+1;j<mass.length;j++)
// если i-тый  и  j-тый элементы равны
if (mass[i]==mass[j]){
// то изменить метки  этих элементов
                                                                    metka[i]= true;
 metka[j] = true;
metka[i] = true;}
//  в зависимости от значений метки определять цвет вывода элемента
for(int i=0;i<mass.length;i++)
{if (metka[i]) G.setColor(Color.RED);
```

else G.setColor(Color.GREEN);
*G.drawString(new Integer(mass[i]).toString(),i*20,40);}}}}*

Лабораторная работа № 16. «Управляющие конструкции»

Лабораторная работа № 17. Массивы и строки

Лабораторная работа № 18 Внутренние классы

Цель работы

Научиться применять на практике принцип наследования объектно-ориентированного программирования с использованием специальных механизмов языка *Java.* Применить на практике теоретический материал, изложенный в разделе «Внутренние и вложенные классы».

Абстрактные классы

Абстрактные классы содержат объявления абстрактных методов, которые не реализованы в этих классах, а будут реализованы в подклассах. Объекты абстрактных классов создать нельзя. Но можно создавать объекты подклассов, реализующих абстрактные методы. Синтаксис объявления абстрактного класса:

abstract class <имя класса>*{*
// абстрактный метод
abstract <тип возвращаемого значения><имя класса>*(*<параметры>*);*
// обыкновенный метод
*void Met(){*реализация*}*
}

Интерфейсы

Интерфейсы представляют собой полностью абстрактные классы, то есть ни один из объявленных методов не может быть реализован внутри интерфейса. Все поля интерфейса автоматически получают атрибуты доступа и спецификаторы *public, static, final,* а все методы как *public* и *abstract.*

Говорят, что класс реализует интерфейс, переопределяя его методы. При этом класс может реализовывать несколько интерфейсов. Если класс переопределяет не все методы интерфейса, он должен быть объявлен как абстрактный.

Синтаксис определения интерфейса:

[public] interface <имя> *[extends I1, I2, …, IN]*
{ реализация интерфейса*}*

По приведенному синтаксису видно, что интерфейс может быть наследником одного или нескольких интерфейсов (для интерфейсов возможно множественное наследование).

Синтаксис реализации интерфейсов классами:

[доступ] *class* <имя класса> *implements I1, I2, …, IN*
{ код класса
}

При этом *I1, I2, …, IN* - перечисление имен реализуемых классом интерфейсов.

Внутренние классы

Нестатические вложенные классы принято называть внутренними (*inner*). Из внешнего класса доступ к элементам внутреннего класса возможен только через объект внутреннего класса. Этот объект должен быть описан в коде внешнего класса. Методы внутреннего класса имеют прямой доступ к полям и методам внешнего класса.

При проектировании необходимо помнить, что во внутреннем классе нельзя определять статические поля и методы.

Внутренние классы могут быть наследниками и потомками других классов, реализовывать интерфейсы.

Наследование может быть организовано двумя способами:

а)

```
class A{
// поля и методы
   [доступ] class B [extends …][implements…]{
  // поля и методы
   }}

class A1 extends A{
    class B1 extends B{}
    // инициализация объекта класса B1
    B obj = new B1()}
```

б)

```
class B2 extends A.B{
```

/* при таком способе наследования класс *B2* не имеет доступа к полям внешнего класса *A,* поэтому в классе *B2* необходимо объявить конструктор с параметром-объектом внешнего класса, после этого, класс *B2* получит ссылку на класс *B* */

```
B2 (A obj){
  obj.super();
}}
```

Продемонстрируем пример объявления ссылок на объекты внутренних классов.

```
class A {
// защищенный внутренний класс
   protected class B {
      private int x;
      void putx(int x){
      this.x = x;}
      void show (){
      System.out.println("x = "+x);}}
// метод внешнего класса, возвращающий объект внутреннего класса
  B get(){
   return new B;
  }
}
class Demo{
public static void main(String [] a){
// создать объект внешнего класса
A obj = new A();
// создать объект внутреннего класса (обратите внимание на синтаксис)
A.B obj1 = obj.get();
// задать значение поля x
```

```
obj1.putx(17);
// вывести на консоль значение поля x
obj1.show();}}
```

Вложенные классы

Если при описании внутреннего класса используется ключевое слово *static,* такой класс называют вложенным (*nested*) классом. Такой класс может обращаться к нестатическим полям и методам класса только через объект внешнего класса. К статическим полям и методам внешнего класса вложенный класс может обращаться напрямую.

```
class A{
  int x,y;
  static float f;
  void metod1(){x++;}
  void putxy(int x, int y){
  this.x = x; this.y = y;}
  static void show(){System.out.println("AB");}
  static class B{
  static void metod(){
  //обращение к нестатическим полям и методам класса A
  A obj = new A();
  obj.x =obj.x+2;
  obj.y = obj.y+1;
  obj.metod1();
  System.out.println("x = "+obj.x+"y = "+obj.y);}
  //обращение к статическим полям класса A
  void metod2(){
  f = 45;
  show();
  System.out.println(f);}}
//вызов нестатического метода вложенного класса
void metod3(){
new B().metod2();}

public static void main(String [] a){
A obj = new A();
obj.putxy(12,15);
obj.metod3();
// вызов статического метода вложенного класса
A.B.metod();}}
```

На консоль выведется:

AB

45.0

x = 3 *y* = 1

Примеры выполнения лабораторной работы

Абстрактный класс

Рассмотрим реализацию следующей заадчи: создать абстрактный класс **Учащийся** и его подклассы **Школьник** и **Студент.** Создать массив объектов абстрактного класса. Показать отдельно студентов и школьников.

```
// реализация абстрактного класса
import java.util.*;
import javax.swing.*;
abstract public class Learn {
String name;// имя
Date bday; // дата рождения
String scool; // название учебного учереждения
// абстрактный метод редактирования полей обекта
abstract void PutInfo(); }
```

В абстрактный класс включены общие для классов **Школьник** и **Студент** поля и методы.

```
// реализация класса Студент
import javax.swing.*;
import java.util.*;
import java.awt.*;
public class Student extends Learn{
String group; // номер группы
// конструктор
Student(String name, Date bday,String scool, String _class){
   this.name = name;
   this.bday = bday;
   this.scool = scool;
   this.group = _class;}
// конструктор без параметров
Student(){
   this.name  = "";
   this.bday  = new Date();
   this.scool = "";
   this.group = "";}
/*  редактирование полей объекта – метод использует объекты класса JOptionPane  и его статический метод ShowInputDialog, который вводит текстовую информацию во всплывающем окне, параметр метода – строка с названием вводимого поля */
void PutInfo(){
name = JOptionPane.showInputDialog("Name: ");
String day = JOptionPane.showInputDialog("Birth day: ");
bday = new Date(day);
scool = JOptionPane.showInputDialog("VUZ: ");
group = JOptionPane.showInputDialog("Group: ");}}
// реализация класса Школьник
import javax.swing.*;
import java.util.*;
import java.awt.*;
public class Scool extends Learn{
String _class; // класс
// конструктор
Scool(String name, Date bday,String scool, String _class){
      this.name = name;
      this.bday = bday;
      this.scool = scool;
      this._class = _class;}
// конструктор без параметров
Scool(){
```

```
this.name  = "";
this.bday  = new Date();
this.scool = "";
this._class = "";}
/* редактирование полей объекта с помощью объектов класса JOptionPane*/
void PutInfo(){
      name = JOptionPane.showInputDialog("Name: ");
      String day = JOptionPane.showInputDialog("Birth day: ");
      bday = new Date(day);
      scool = JOptionPane.showInputDialog("Scool: ");
      _class = JOptionPane.showInputDialog("Class: ");}}
//  управляющий класс
import javax.swing.*;
import java.awt.*;
import java.util.*;
public class MyFrame extends JFrame{
Object [][] cells;
/* конструкор, параметрами которого являются данные об объектах (Object [][]c), строка с названиями столбцов (String []cN), строка заголовка (String T) */
MyFrame(Object [][]c, String []cN, String T){
setSize(new Dimension(500,400)); // установить размер окна
setTitle(T); // установить заголовок
setBackground(Color.WHITE); // установить цвет фона
setForeground(Color.BLUE); // установить цвет символов
int p = c.length/4;
 cells =new Object[c.length][4];
//  переписать данные об объекте в поле cells
for(int i=0;i<cells.length;i++)
{cells[i][0]=c[i][0];
cells[i][1]=((Date)c[i][1]).getDay()+"."
+((Date)c[i][1]).getMonth()+"."+
 ((Date)c[i][1]).getYear();
cells[i][2]=c[i][2];
cells[i][3]=c[i][3];}
JTable table = new JTable(cells,cN);//  создать таблицу
getContentPane().add(new JScrollPane(table), // добавить ее во фрейм
BorderLayout.CENTER); //  расположить ее по центру
} public static void main(String[] args) {
// TODO Auto-generated method stub
Learn [] S = new Learn[4]; // создать массив объектов Учащийся
/*создать диалоговое окно со списком, для этого определить возможные варианты выбора в переменной possibleValues */
Object [] possibleValues = {
"student",
"scool"};
int k = 0, k1 = 0;
/* для каждого вводимого объекта с помощью списка определить, к какому классу относится объект, для этого создается диалоговое окно с указанными параметрами: */
for (int i=0;i<S.length;i++)
{ Object selectedValue =
JOptionPane.showInputDialog(
null, // родительский объект
```

```
"Choice type: ", // текстовая метка в окне диалога
"Choice", // заголовок
JOptionPane.INFORMATION_MESSAGE, // тип сообщения
null, // иконка в окне диалога (null – по умолчанию)
possibleValues, // значения элементов списка
possibleValues[0]); // начальное значение
// если выбран объект Школьник
  if ((String)selectedValue == "scool")
  {S[i] = new Scool();// выделить память
  S[i].PutInfo(); // заполнить данные
  k++;}
else { // в противном случае создается и заполняется объект Студент
S[i] = new Student();
  S[i].PutInfo();
  k1++;}}
/* переписать данные в массивы cells и cells1
Object [][] cells = new Object[k][4];
Object [][] cells1 = new Object[k1][4];
for(int i=0,j=0,z=0;i<S.length;i++){
if (S[i] instanceof Scool){
cells[j][0] = S[i].name;
cells[j][1] = S[i].bday;
cells[j][2] = S[i].scool;
cells[j][3] = ((Scool)S[i])._class;
j++;}
else {
cells1[z][0] = S[i].name;
cells1[z][1] = S[i].bday;
cells1[z][2] = S[i].scool;
cells1[z][3] = ((Student)S[i]).group;
z++;}}
// заголовки столбцов таблицы для Школьников
String _scool [] = {
"Имя",
"Дата рождения",
"Школа",
"Класс"
};
// заголовки столбцов таблицы для Студентов
String _student [] = {
"Имя",
"Дата рождения",
"ВУЗ",
"Группа"};
// Создать фрейм, содержащий таблицу со Школьниками
MyFrame a = new MyFrame(cells,_scool,"Школьники");
a.setVisible(true);
// Создать фрейм, содержащий таблицу со Студентами
MyFrame b = new MyFrame(cells1,_scool,"Студенты");
b.setVisible(true);}}
```

Интерфейс

Создать интерфейс **Фигура** и его реализации для классов ***Окружность*** и **Квадрат.** Предусмотрим в интерфейсе методы вычисления площади фигуры и методы рисования.

```
import java.awt.*; // интерфейс
public interface Shape {
double pi = Math.PI;
public double Square();
public void Show (Graphics g);}

// класс окружность – реализация интерфейса
public class Circle implements Shape{
 int x,y,rad;
 Circle(int x, int y, int rad){ // конструктор
      this.x=x;
     this.y=y;
      this.rad=rad; }
 public double Square(){ // вычисление площади круга
      return 2*pi*rad*rad; }
 public void Show (Graphics g){//  рисование окружности
      g.drawOval(x,y,rad,rad); }}
// класс квадрат – реализация интерфейса
public class Quadro implements Shape{
int x,y,rad;
Quadro(int x, int y, int rad){ // конструктор
       this.x=x;
       this.y=y;
       this.rad=rad;}
 public double Square(){ // вычисление площади
       return 4*rad*rad; }
 public void Show (Graphics g){ // рисование
       g.drawRect(x,y,rad,rad); }}

import javax.swing.*; //  класс для работы с фреймом
import java.awt.*;
public class FrameI extends JFrame{
 Shape c1,c2; // описание ссылок на интерфейс
 FrameI(){ // конструктор
      this.setSize(new Dimension(500,500)); // установить размер,
      this.setBackground(Color.gray);          // цвет фона
      this.setForeground(Color.YELLOW);  // цвет рисования,
      this.setTitle("Shape");                    // заголовок
     c1 = new Circle(50,50,25);                // создание окружности
     c2 = new Quadro(50,140,25); }          // создание квадрата
 public void paint(Graphics g){
      double sq = c1.Square(); // расчет площади окружности
      double sq1 = c2.Square();// расчет площади квадрата
      c1.Show(g); // нарисовать окружность
      c2.Show(g);// нарисовать квадрат
     g.drawString("Square " + sq,25,100);     //  вывести значение
     g.drawString("Square " + sq1,25,195); }// площадей
 public static void main(String [] a){
```

```
    FrameI Fr = new FrameI(); // создать фрейм
    Fr.setVisible(true); // сделать его видимым
   Fr.repaint(); // вызвать метод рисования
}}
```

Внутренний класс

Создать класс Записная книжка, с внутренним классом или классами, с помощью объектов которого могут храниться несколько записей на одну дату.

```
package p4;
import java.util.*;
import javax.swing.*;
public class Notepad {
int m;  // количество записей по датам
int n;  // количество записей с телефонами
Note [] N; // массив записей по датам
Telephone [] T; // массив записей с телефонами
// реализация класса «Запись по датам»
class Note {
Date d; //  дата
String [] note; // массив записей
int m; // количество записей
Note(Date d){ // конструктор
this.d = d;
note = new String [20];
m = 0;}
// создать новую запись
void SetNote(String str){
note[m]=str;
m++;}
Object []GetInfo(){// вернуть запись
Object [] c = new Object[m+1];
c[0]=d.getDay()+"."
      +(d.getMonth()+1)+"."+
       (d.getYear()+1900);
for(int i=0;i<m;i++)
 {c[i+1] = note[i];} return c;} }
// реализация класса «Запись телефона»
class Telephone {
String name;
String telephone;
void SetTelephone(String n, String t){ // создание новой записи
if (n!="") name = n;
if (t!="") telephone = t;}
Object [] GetInfo(){ // вернуть запись
Object [] c = new Object[2];
c[0] = name;
c[1] = telephone;
return c;}}
 Notepad(){ // конструктор
 m = 0; // количество записей обнуляется
```

```
 n = 0;
N = new Note [30];
T = new Telephone[30]; }
// создать новую запись по дате
 void SetNote(){
 boolean flag = false;
 String day = JOptionPane.showInputDialog("Date: ");
Date d = new Date(day);
 String n = JOptionPane.showInputDialog("Note: ");
 if (m==0){ N[m]=new Note(d); // если записная книжка пуста
              N[m].SetNote(n);
               m++; }
 else // сравнение веденной даты с уже имеющимися
 for(int i=0;i<m;i++)
 {if (d.equals(N[i].d)) {flag = true;
                          N[i].SetNote(n);
                          break;}
 else {
 if (!flag) {
 N[m]=new Note(d);
 N[m].SetNote(n);
 m++;
 break; }} } }
// ввод новой записи с телефоном
 void SetTelephone(){
boolean flag = false;
String name = JOptionPane.showInputDialog("Name: ");
 String t = JOptionPane.showInputDialog("Telephone: ");
 for(int i=0;i<n;i++){ // сравнение телефона с уже имеющимися
if (name.equals(((Telephone)T[i]).name)&&
!T.equals(((Telephone)T[i]).telephone)){
 int result = JOptionPane.showConfirmDialog(
 null,
 new String("Edit "+name),
 "Attention!",
 JOptionPane.YES_NO_OPTION);
 if (result==0) {T[i].SetTelephone("",t); flag = true; break;} }
 if (!name.equals(((Telephone)T[i]).name)&& // сравнение фамилии с
// уже имеющимися
T.equals(((Telephone)T[i]).telephone))
 {int result = JOptionPane.showConfirmDialog(
 null,
 new String("Edit "+t),
 "Attention!",
 JOptionPane.YES_NO_OPTION);
 if (result==0) {T[i].SetTelephone(name,""); flag = true; break;}}
 if (name.equals(((Telephone)T[i]).name)&&
T.equals(((Telephone)T[i]).telephone))
{int result = JOptionPane.showConfirmDialog(
 null,
 new String("Edit "+name + " "+t+ " ?"),
 "Attention!",
```

```
 JOptionPane.YES_NO_OPTION);
 if (result==0) {T[i].SetTelephone(name,t); flag = true; break;}} }
 if (!flag){T[n] = new Telephone();
  T[n].SetTelephone(name,t);
  n++;} }
Object [][] GetInfoT(){// вернуть информацию о телефонах
 Object [][] c = new Object[n][2];
for (int i=0;i<n;i++)
 {c [i]= T[i].GetInfo(); }
 return c; }
 Object [][] GetInfoN(){ // вернуть информацию о записях
Object [][] c = new Object[this.GetM()][2];
 int k = 0;
for (int i=0;i<m;i++)
 {   Object [] c1 = new Object[N[i].m+1];
 c1 = N[i].GetInfo();
for(int j=k,j1 = 1;j1<c1.length;j++,k++,j1++){
 c[j][0]=c1[0];
 c[j][1]=c1[j1];}}
 return c; }
 int GetN(){
return n; }
 int GetM(){
 int z = 0;
for(int i=0;i<m;i++){
 z+=N[i].m; }
 return z; }}

// исполняемый класс
package p4;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.util.Date;
import java.awt.event.*;
import javax.swing.*;
public class FrameN extends JFrame implements ActionListener{
JMenu menN;JMenu menT;JMenuItem TE;
JMenuItem NE;JMenuItem TS;JMenuItem NS;
JMenuItem E;
Notepad notepad;
FrameN(Object [][]c, String []cN, String T){// конструктор,
// использующийся для создания фрейма с таблицами
setSize(new Dimension(500,400));
setTitle(T);
setBackground(Color.WHITE);
setForeground(Color.BLUE);
JTable table = new JTable(c,cN);
getContentPane().add(new JScrollPane(table),
 BorderLayout.CENTER);}
FrameN(){// конструктор, использующийся для создания фрейма
// с меню
```

```
notepad = new Notepad();
setSize(new Dimension(500,400));
setTitle("Notebook");
setBackground(Color.WHITE);
setForeground(Color.BLUE);}
public void init(){ // инициализация
menT = new JMenu("Telephone"); // создание двух пунктов меню
menN = new JMenu("Note");
TE = new JMenuItem("New");// создание пунктов подменю
NE = new JMenuItem("New");
TS = new JMenuItem("Show");
NS = new JMenuItem("Show");
E = new JMenuItem("Exit");
JMenuBar menubar = new JMenuBar();
this.setJMenuBar(menubar);// создание поля для меню на фрейме
TE.addActionListener(this);// добавление пунктов подменю
NE.addActionListener(this);// в блок считывания событий
TS.addActionListener(this);
NS.addActionListener(this);
E.addActionListener(this);
menT.add(TE);menT.add(TS);// добавление пунктов подменю в
menT.add(E);menN.add(NE); // меню
menN.add(NS);
menubar.add(menT);
menubar.add(menN);}
public void actionPerformed(ActionEvent e){ //выбор действий
// определение заголовков таблиц
String [] _n = {"Date",
"Note"};
String [] _t = {"Name",
"Telephone"};
JMenuItem jm = (JMenuItem)e.getSource();
if (jm.equals(TE) ) notepad.SetTelephone();// ввод новой записи с
// телефоном
else if (jm.equals(NE) )notepad.SetNote();// ввод новой записи по дате
else if (jm.equals(TS)){
int n = notepad.GetN();// формирование и вывод таблицы с записями по
Object [][] cells1 = new Object[n][2]; // дате
cells1 = notepad.GetInfoT();
FrameN b = new FrameN(cells1,_t,"TB");
b.setVisible(true);
} else if (jm.equals(NS)){ // формирование и вывод таблицы с записями
int m = notepad.GetM();// телефонов
Object [][] cells = new Object[m][2];
cells = notepad.GetInfoN();
FrameN a = new FrameN(cells,_n,"NB");
a.setVisible(true);}
else if (jm.equals(E)) System.exit(0); // выход
}
public static void main(String[] args) {
FrameN f_menu = new FrameN(); // инициализация фрейма с меню
f_menu.init();
```

```
f_menu.setVisible(true);}}
```

Вложенный класс

Опишем класс «Лучи, с началом в одной точке координат» и вложенный класс «Линия». Объект класса Линия в этом случае, можно создавать напрямую, не используя объект внешнего класса.

```
package p44;
import java.awt.*;
public class Luch { // Класс «Лучи»
   static int x,y; // координаты начала лучей
   int n; // количество лучей
   int [][] xy; // координаты концов лучей
   Line [] l;  // массив объектов вложенного класса «Линия»
  Luch(int n1, int k, int p){ // конструктор класса
   n = n1;
   x = k;
   y = p;
   xy = new int [n][2];
   for (int i=0;i<n;i++) // случайное задание концов лучей
    for(int j=0;j<2;j++)
         xy[i][j] = (int)(1000*Math.random()/10);
    l = new Line[n];
for(int i=0;i<n;i++) // создание объектов класса «Линия»
l[i]=new Line(x,y,xy[i][0],xy[i][1],Color.blue);}
void Show(Graphics g){ // рисование
g.drawOval(x,y,5,5);
for(int i=0;i<n;i++)
  l[i].Show(g);  }
   static class Line { // вложенный класс
int px,py,px1,py1; // координаты концов линии
Color color; // цвет линии
Line(int x1,int y1, int x2, int y2, Color c){// конструктор
px = x1;
py = y1;
px1 = x2;
py1 = y2;
color = c;}
void Show(Graphics g){ // рисование
g.setColor(color);
g.drawLine(px,py,px1,py1);}}}
// описание фрейма для рисования лучей
package p44;
import java.awt.*;
import javax.swing.*;
public class FrameLuch extends JFrame{
Luch luch; // объект класса «Лучи»
FrameLuch(){ // конструктор
this.setTitle("Luch");
this.setBackground(Color.gray);
this.setSize(new Dimension(100,200));
luch = new Luch(10,50,100);}
```

```
public void paint(Graphics g){
luch.Show(g);}}
// описание фрейма для рисования линии
package p44;
import java.awt.*;
import javax.swing.*;
public class FrameLine extends JFrame{
Luch.Line line;
FrameLine(){
this.setTitle("Line");
this.setBackground(Color.gray);
this.setSize(new Dimension(100,200));
line = new Luch.Line(10,10,80,90,Color.yellow);}
public void paint(Graphics g){
line.Show(g);}}
// исполняемый класс
package p44;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FrameL extends JFrame implements ActionListener{
  JMenu menuLuch; // описание меню и пунктов меню
  JMenu menuLine;
 JMenuItem LuchS;
JMenuItem LineS;
JMenuItem E;
FrameL(){ // конструктор
setTitle("Nested class");
setBackground(Color.WHITE);
setForeground(Color.BLUE);
setSize(new Dimension(200,200));
menuLuch = new JMenu("Luch");
menuLine = new JMenu("Line");
LuchS = new JMenuItem("Show");
LineS = new JMenuItem("Show");
E = new JMenuItem("Exit");
JMenuBar menubar = new JMenuBar();
this.setJMenuBar(menubar);
LuchS.addActionListener(this);
LineS.addActionListener(this);
E.addActionListener(this);
menuLuch.add(LuchS);
menuLuch.add(E);
menuLine.add(LineS);
menubar.add(menuLuch);
menubar.add(menuLine);   }
public void actionPerformed(ActionEvent e){// обработка событий
JMenuItem jm = (JMenuItem)e.getSource();
if (jm.equals(LuchS)) {
FrameLuch FL = new FrameLuch(); // выбор рисования лучей
FL.repaint();
```

```
FL.setVisible(true);
}else if (jm.equals(LineS)){// выбор рисования линии
FrameLine fl = new FrameLine();
fl.repaint();
fl.setVisible(true);
}else if (jm.equals(E))System.exit(0);// выход из меню}
public static void main(String[] args) {
        FrameL Fr = new FrameL();
        Fr.setVisible(true);}}
```

Порядок выполнения работы

- Получите индивидуальное задание у преподавателя.
- Разработайте структуру классов по индивидуальному заданию.
- Реализуйте по разработанной структуре графическое приложение на языке *Java.*
- сохраните выполненную работу в *jar* архиве.

Лабораторная работа № 19. «Абстрактные классы и интерфейсы. Наследование и полиморфизм. Создание класса наследника»

Цель работы

Научиться моделировать системы классов, связанные между собой различными отношениями. Практически применить теоретический материал разделов «Отношения между объектами» и «Реализация наследования и полиморфизма в языке *Java*»

Отношения между объектами

Отношением называется связь между объектами. Можно выделить четыре типа отношений:

- Зависимость.
- Обобщение.
- Ассоциация
- Реализация.

Зависимость – отношение использования, определяющее, что изменение состояния одного объекта может повлиять на совершенно другой объект, который его использует. Обратное утверждение неверно. Зависимости применяются тогда, когда один объект использует другой объект (например, в качестве параметра метода).

Отношением зависимости, например, связаны объекты *Пользователь* и *Администратор. Администратор* имеет право изменять имя, пароль и права конкретного *Пользователя.*

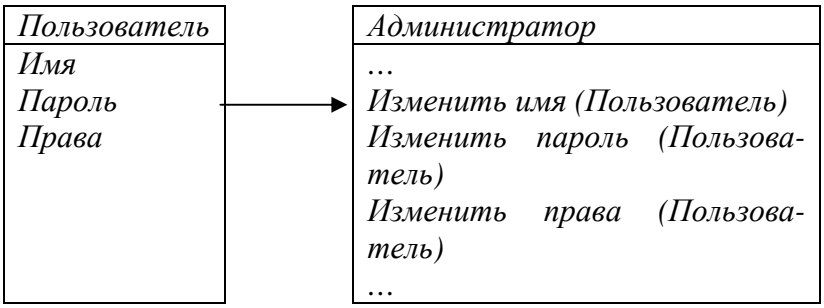


Рис. 1.1. Отношение «Зависимость»

Обобщение – объекты-потомки могут использоваться всюду, где встречаются базовые объекты. Но не наоборот. Отношение обобщения распространяется на объекты, связанные наследованием. Например, отношением обобщения связаны объекты *Врач, Хирург, Нейрохирург*. Базовым объектом в этой схеме, обобщающим свойства всех остальных объектов будет объект *Врач.* Объекты *Хирург* и *Нейрохирург* также связаны между собой отношением обобщения.

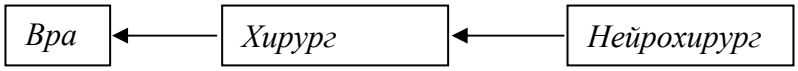


Рис. 1.2. Отношение «Обобщение»

Реализацией – называется отношение между объектами (классификаторами), при котором один из них описывает интерфейс сущности (контракт), а другой гарантирует его выполнение. Сущность, это объект, который в большинстве случаев инкапсулирует в себе только свойства и имеет минимальные права по изменению своих свойств. Примером реализации может быть связь между объектами *Кассир* и *Чек.* У объекта *Кассир* существует метод, который создает объект *Чек.* Или реализует объект *Чек.*

Ассоциация – это отношение, при котором объекты одного типа связаны с объектами другого и отражают некоторое отношение между ними. При этом оба класса при этом находятся на одном концептуальном уровне иерархической схемы. Пример ассоциативного отношения приведен на рис. 1.3.

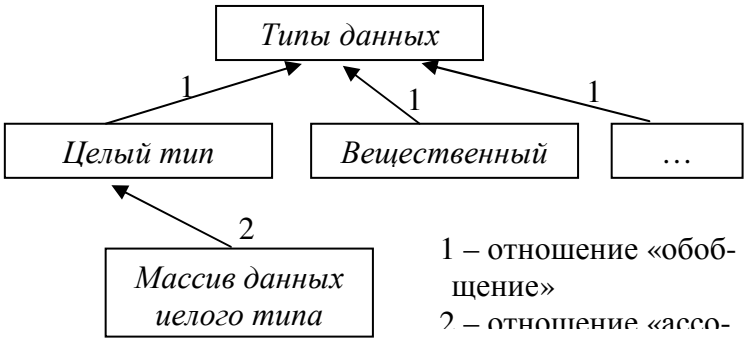
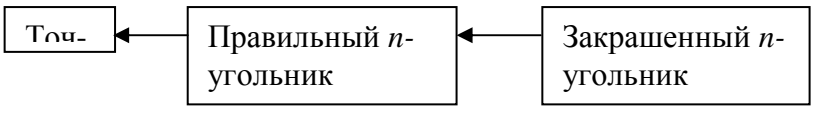


Рис. 1.3. Отношение «Ассоциация»

Наследование

Наследование - это способность брать существующий объект, будем называть его базовым, и порождать от него новый объект - потомок, с наследованием всех его свойств и поведения. Приведем следующий пример наследования:



Объект *Точка* описывается координатами и его координаты можно изменить. Объект *Правильный n-угольник* наследует координаты (определим их как координаты центра), и

свойство изменять координаты. От *Точки Правильный n-угольник* отличается своим собственным полем *радиус вписанной окружности* и целым числом *n* – количество углов. Объект *Закрашенный n-угольник* является потомком *Правильного n-угольника* и приобретает новое свойство – *цвет.*

У объекта может быть несколько потомков. Пример такой иерархической схемы приведен на рис. 1.4.

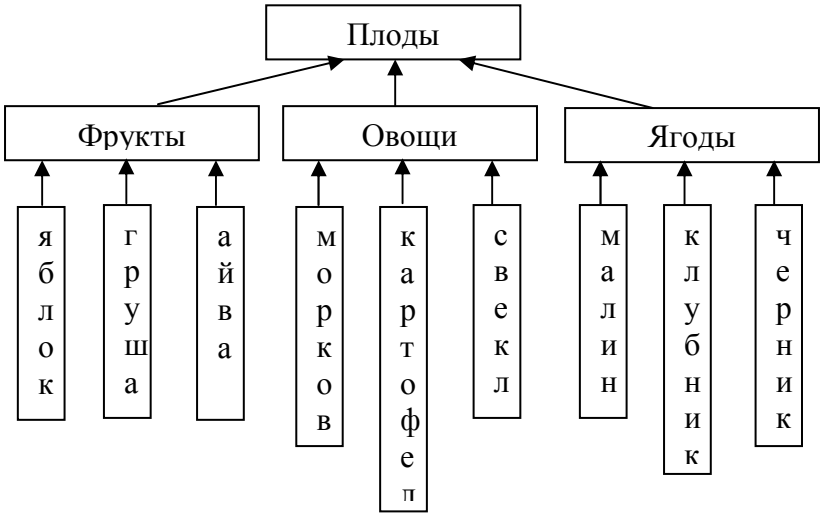


Рис. 1.4. Пример иерархии наследования

Синтаксис объявления класса-наследника в *Java:*

class A { …}
class B extends A{…}.

Говорят, что класс *B* – расширение класса *A,* или наследник класса *A,* или производный класс от *A,* или потомок класса *A.*

Приведем пример наследования

Напишем класс «Точка», его наследник класс «Правильный *n*-угольник» и фрейм для демонстрации работы с классами. Фрейм – пользовательское графическое приложение, наследник класса *JFrame* из библиотеки *javax.swing.*

```
import java.awt.*;
import javax.swing.*;
public class Pixel {
    int x,y; // координаты центра
    Color c; // цвет
    Pixel(int x,int y,Color c){ // конструктор, задающий поля объекта
       this.x = x;              // из параметров
       this.y = y;
       this.c = c;}
    Pixel(){ // конструктор без параметров
       x = 0;
       y = 0;
       c = Color.BLACK;}
```

```
    void show(JFrame a, Graphics g){ // рисование объекта заданным
                                     // цветом
      g.setColor(c);
      g.drawOval(x,y,3,3);}
    void hide(JFrame a, Graphics g){ // стирание объекта
      a.setForeground(a.getBackground());
      g.drawOval(x,y,3,3);}}
```

// наследник класса «Точка», класс «Правильный *n*-угольник»

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.*;
import javax.swing.JFrame;
public class Poly extends Pixel{
// поля x,y,c – не описываются, считаются унаследованными из базового
///класса
int n,r; // количество углов и радиус описанной окружности
Poly(int x, int y, Color c, int n, int r){ // конструктор
super(x,y,c); // обратите внимание – обязательный вызов конструктора
// супер-класса
this.n = n;
this.r = r; }
// рисование n-угольника
void show(JFrame a, Graphics g){
g.setColor(c);
int x1 = x;
int y1 = y-r;
int y2,x2;
for(int i=1;i<=n;i++){
double x11 = Math.sin(2*i*Math.PI/n)*r;
x2 = x+(int)x11;
double y11 = Math.cos(2*i*Math.PI/n)*r;
y2 = y-(int)y11;
g.drawLine(x1,y1,x2,y2);
x1 = x2;
y1 = y2;}}
// стирание n-угольника
void hide(JFrame a, Graphics g){
g.setColor(a.getBackground());
int x1 = x;
int y1 = y-r;
int y2,x2;
for(int i=1;i<=n;i++){
double x11 = Math.sin(2*i*Math.PI/n)*r;
x2 = x+(int)x11;
double y11 = Math.cos(2*i*Math.PI/n)*r;
y2 = y-(int)y11;
g.drawLine(x1,y1,x2,y2);
x1 = x2;
y1 = y2;}}}
 // класс, создающий фрейм – графическое приложение пользователя
```

```
import javax.swing.*;
import java.awt.*;
// все фреймы, создаваемые пользователем – наследники класса JFrame
// (или Frame)
public class MyFrame extends JFrame{
// поля класса – объекты классов Pixel
Pixel z,m;
// конструктор класса, инициализирующий поля класса указанным
// образом
MyFrame(){
z = new Pixel(40,40,Color.CYAN);
m = new Poly(100,100,Color.YELLOW,5,25);
}
// рисование объектов, в качестве параметров метода show используется
// указатель на создаваемый фрейм и графическая среда (параметр
// метода paint)
public void paint(Graphics g){
        g.clearRect(0, 0, getBounds().width, getBounds().height);
      z.show(this,g);
      m.show(this,g);}
// Для запуска фрейма необходим метод main
public static void main(String[] args) {
// создать объект класса MyFrame
MyFrame a = new MyFrame();
// установить его размеры
a.setSize(new Dimension(500,400));
// установить заголовок
a.setTitle("Example 1");
// установить цвет фона
a.setBackground(Color.darkGray);
// сделать объект видимым – обязательно!
a.setVisible(true);
// вызвать метод paint, обязательно через repaint
a.repaint();}}
```

Полиморфизм

Полиморфизм – это способность изменять свойства объектов-потомков, не изменяя названий этих свойств. В языке *Java* существует только динамический полиморфизм. Поэтому в рассмотренном выше примере поля *m* и *z* класса *MyFrame* объявлены как объекты базового класса, память под эти объекты выделялась различными способами и при вызове метода *show* виртуальная машина самостоятельно определяет какой из методов необходимо выполнить. При этом *Java* ориентируется именно на инициализацию, а не на описание.

Создание исполняемых jar-архивов

Выполнение *java*-приложений возможно не только из среды *Eclipse.* Возможно создание выполняемого архива с расширением *jar.* Среда *Eclipse* предоставляет инструментарий для создания таких архивов. Для этого нужно выполнить следующие действия –

- выбрать пункт меню *File/Export;*
- выбрать место экспортирования файлов – *JAR file;*

- в диалоговом окне выбрать файлы для экспортирования и место создания *jar*-архива, на рис. 1.5. для экспортирования выбран пакет *p3* проекта *Met, jar*-архив будет создаваться в папке *D:\workspase* и будет называться *MyFrame;*
- перейти к следующему этапу экспорта, определяющему опции упаковки файлов – упаковка файлов с ошибками компиляции и с предупреждениями и определение файла для хранения отчета;
- перейти к следующему (завершающему) шагу – автоматическое создание файла *manifest,* на этом шаге необходимо указать класс с методом *main(),* при этом на экран выводится диалоговое окно с указанием классов пакета, имеющих метод *main(),* для рассматриваемого примера – *MyFrame (*рис. 1.6.*)*

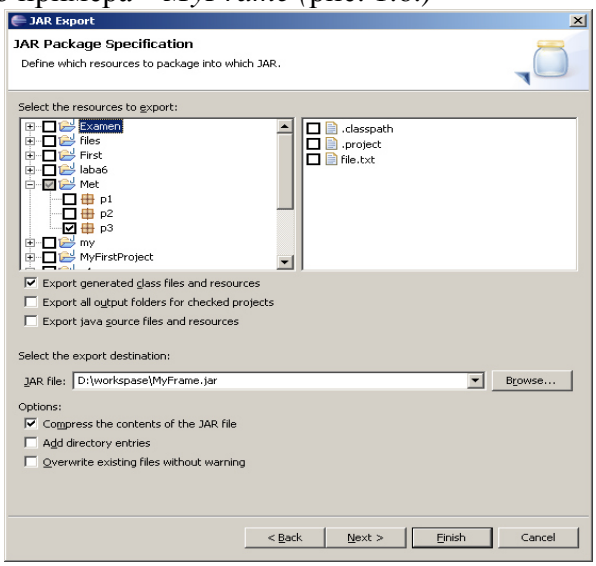


Рис. 1.5. Диалоговое окно «Выбор ресурсов для экспорта»

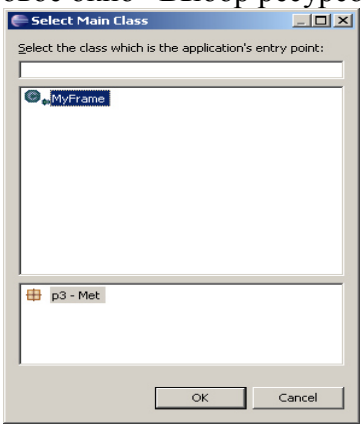


Рис. 1.6. Диалоговое окно «Выбор главного класса»

Лабораторная работа №21-22. Файлы. Обработка исключительных ситуаций

Цель работы

Целью данной работы является изучение классов *Java,* использующихся при работе с файлами, способов обработки исключительных ситуаций, возникающих во время работы программы и способов хранения и обработки объектов.

Файлы. Потоки ввода-вывода

Для работы с файлами в приложениях *Java* используются классы из пакета *java.io.* Класс *File* служит для хранения и обработки в качестве объектов и каталогов и имен файлов. Этот класс работает со следующими свойствами файлов:

- права доступа;
- дата и время создания;
- полный путь к файлу;
- создание и удаление файла;
- изменение имени файла или каталога и т.д..

Класс *File* содержит следующие поля и методы:

static String pathSeparator - определенный системой символ разделитель, используйщийся в пути к файлу, преобразованный в тип *String.*
static char pathSeparatorChar – символ-разделитель, сохраненный в типе *char.*
static String separator – определенный системой разделитель, использующийся в именах файлов, преобразованный к типу *String.*
static char separatorChar – символ-разделитель, сохраненный в типе *char.*

Конструкторы:
File(String pathname) – связывает объект с указанным файлом.
File(File parent, String child) – связывает объект с файлом, который находится в каталоге *parent* и имеет имя *child.*
File(String parent, String child) – связывает объект с файлом, который находится в каталоге *parent* и имеет имя *child.*

Методы:
boolean canRead() – тестирование на возможность чтения.
boolean canWrite() – тестирование на возможность записи.
int compareTo() – сравнение двух объектов по имени (сравнение проходит в лексикографическом порядке).
boolean createNewFile() – создает новый пустой файл с заданным именем, только если этот файл не существует.
static File createTempFile(String prefix,String suffix, File directory) – создает временный файл с заданным префиксом и суффиксом в заданной директории.
boolean delete() – удаление файла.
bolean deleteOnExit() – удаление при выходе из приложения.
boolean exists() – проверка существования.
File getAbsoluteFile – возвращает абсолютную форму записи пути в переменную типа *File.*
String getAbsolutePath() - возвращает абсолютную форму записи пути в переменную типа *String.*
File getCanonicalFile – возвращает каноническую форму записи пути в переменную типа *File.*
String getCanonicalPath() - возвращает каноническую форму записи пути в переменную типа *String.*
String getPath() – преобразует абстрактный путь к строковому типу.
boolean isAbsolute() – сравнение абстрактного имени с абсолютным.
boolean isDirectory() – тестирование, является ли объект директорией.
boolean isFile() – тестирование, является ли объект файлом.
boolean isHidden() – тестирование, является ли объект скрытым файлом.
long lastModified() – возвращает время последнего изменения объекта.
long length() – возвращает длину в байтах существующего файла.
String[] list() – возвращает список директорий и файлов в объекте-директории.
String[] list(FilenameFilter filter) – возвращает список файлов и директорий по указанному шаблону.
File[] listFiles() – возвращает список объектов в объекте-директории.

File[] listFiles(FilenameFilter filter) – возвращает список объектов в объекте-директории по указанному шаблону.
`static` *File*`[]` *listRoots*`()` – возвращает список доступных дисков.
boolean mkdir() – создает директорию с указанным именем.
boolean renameTo(File dest) – переименовывает объект в имя dest.
`boolean setLastModified(long time)` – устанавливает время последнего изменения файла.
`boolean setReadOnly()` – устанавливает для объекта атрибут «только для чтения».

Потоки ввода-вывода последовательности байт являются подклассами абстрактного класса *InputStream* и *OutputStream.* При работе с файлами используются классы *FileInputStream, FileOutputStream.* Конструкторы этих файлов открывают поток и связывают его с соответствующим файлом. Так же, для ввода-вывода потока байт можно использовать объекты классов *FileReader* и *FileWriter.*

Для преобразования введенных данных к базовым типам библиотека ввода-вывода содержит классы *DataInputStream* и *DataOutputStream.* Для этого используются методы *readBoolean()(writeBoolean()), readByte(writeByte()), readInt()(writeInt())* и т.д..

Все объекты выше описанных классов осуществляют последовательный доступ к информации. Прямой доступ к информации можно получить с помощью объектов класса *RandomAccessFile.* Конструктор класса *RandomAccessFile(File f, String mode)* связывает поток с файлом *file* и устанавливает режим доступа *mode.* Параметр *mode* равен *“r”* для чтения и *“wr”* для записи. Для создания потока прямого доступа можно использовать и другой конструктор – *RandomAccessFile(String name, String mode),* где *name* – имя файла.

Ввод-вывод информации в текстовом виде можно осуществить, используя предопределенные потоки. Поле *in,* содержащееся в классе *System* и определенное как консоль можно переназначить на другое устройство. При этом для ввода используется подкласс *BufferedReader* и методы *read()* и *readLine()* для чтения символа и строки. Для вывода текстовой информации используется класс *PrintWriter* и его методы *print()* и *println().*

В библиотеке ввода-вывода *Java* есть классы, позволяющие вводить и выводить объекты других классов. Перед операциями ввода-вывода объекты необходимо преобразовать в поток байт. Это действие называется сериализацией. Для проведения сериализации объекта класса, необходимо, чтобы класс расширял интерфейс *Serializable.* Этот интерфейс не имеет методов и может сериализовать все поля класса, не имеющие спецификаторов *static* и *transient.* Для вывода объектов используется класс *ObjectOutputStream (*метод *writeObject(Object o))*, для чтения – класс *ObjectInputStream (*метод *readObject(Object o)).*

Обработка исключительных ситуаций

Исключительные ситуации возникают во время выполнения программы, при этом возникшая проблема не может быть разрешена в текущем контексте. При возникновении исключения создается объект, который описывает это исключение. Ссылка на этот объект передается обработчику исключений, который пытается решить эту проблему. Если в коде программы используется метод, в котором может возникнуть исключительная ситуация, но не предусмотрена ее обработка, ошибка возникает еще на этапе компиляции.

Исключения являются наследниками класса *Throwable.*Каждой исключительной ситуации поставлен в соответствие некоторый класс. Например: *ClassNotFoundException* – не найден класс, *NoSuchFieldException* – нет такого поля и т.д..

Существует три способа обработки исключений:

- перехват и обработка исключений в блоках *try-catch;*
- объявление исключений в секции *throws* метода и передача вызывающему методу;
- перехват исключения, преобразование его к другому классу и повторный вызов.

Операторы *try* можно вкладывать друг в друга.

Оператор *throw* используется для генерации исключения. Синтаксис оператора:

throw объект*Throwable*;

При достижении этого оператора выполнение кода прекращается, управление передается ближайшему блоку *try-catch.*

Оператор *throws* указывает на то, что метод отказывается от обработки исключения.

Если в программе необходимо выполнить какие-либо действия, независимо от того, произошло исключение или нет, используется блок *finally.*

Пример выполнения лабораторной работы

Реализовать графическое приложение на языке *java* – в текстовом файле записаны элементы целочисленного массива. Найти количество элементов массива, значения которых не уникальны. На экран вывести найденное значение и значения повторяющихся элементов. Выполнить обработку исключительных ситуаций.

При решении данной задачи могут возникнуть следующие исключительные ситуации – файл не найден, файл содержит некорректные данные (данные, не являющиеся целыми числами). Напишем два класса, являющиеся наследниками класса *Exception* – *NotFound* и *ErrorData.*

```
import javax.swing.*;
public class NotFound extends Exception{
 public NotFound(String e){
     super(e); } // Конструктор вызывает конструктор суперкласса
public void Warning(){ // формирование окна сообщения
 JOptionPane.showMessageDialog(null,
                                                    "File     not
found"); }}
import javax.swing.*;
public class ErrorData extends Exception{
 public ErrorData(String e){
 super(e); } // Конструктор вызывает конструктор суперкласса
 public void Warning(){
JOptionPane.showMessageDialog(null,
                    "Wrong data"); }}
```

Для работы с массивом описан класс *Massiv.* Конструктор класса инициализирует массив – объект класса *ArrayList.* Для решения поставленной задачи в методе *Metod* воспользуемся свойствами класса *HashSet.* Опишем два объекта-множества – *set* и *set1.* Во множество *set1* будем добавлять элементы по мере их поступления из массива *m.* Элементы, значения которых уже содержатся в *set1,* будем записывать в *set.* После просмотра всего массива *m* во множестве *set* будут находиться элементы, значения которых повторяются в исходном массиве.

```
import java.util.*;
public class Massiv {
ArrayList m;
Massiv (ArrayList s){
 m = new ArrayList();
 m.addAll(s); }
HashSet Metod(){
 HashSet set = new HashSet();
 HashSet set1 = new HashSet();
 for(int i=0;i<m.size();i++){
     if(!set1.contains(m.get(i)))set1.add(m.get(i));
       else set.add(m.get(i));  }
 return set;  }}
```

Для вывода информации на экран напишем класс *DataWin,* расширяющий класс *JFrame.* Конструктор этого класса создает окно, содержащее заданную информацию. Размеры этого окна изменять нельзя, однако можно выполнять просмотр нередактируемой информации. Информация, отображающаяся в окне – объект класса *Collection.* Объект класса *DataWin* будет использоваться и для просмотра исходного массива и для просмотра множества повторяющихся элементов.

```
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.*;
public class DataWin extends JFrame{
 TextArea textarea; // Текстовое поле
 DataWin(Collection data, String title){
  addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
              setVisible(false);
              dispose();   };    });
      textarea = new TextArea(10,10);
      textarea.setEditable(false);
      Iterator it = data.iterator(); // извлечение итератора
      // просмотр коллекции
   while(it.hasNext())
      { textarea.append(it.next().toString()+"\n");      }
      setSize(new Dimension(100,140));
      setLocation(320,100);
      setTitle(title);
      setResizable(false);
      add(textarea);
      setVisible(true); }}
```

Класс *ReadFile* является выполняемым классом. Метод класса *Read* запрашивает имя текстового файла, выполняет перехват исключительных ситуаций, описанных в классах *NotFound* и *ErrorData.* В случае успеха метод считывает информацию в строковую переменную. С помощью статических методов класса *StringTokenizer* выделяет лексемы исходной строки и преобразует их к целому типу. Полученные числа сохраняются в коллекции *ArrayList.*

```
import java.io.*;
import java.util.*;
import javax.swing.JOptionPane;
public class ReadFile {
String name;
ArrayList  Read(){
String str = "";
ArrayList x = new ArrayList();
name = JOptionPane.showInputDialog("Enter file name: ");
try{ try{
      BufferedReader file = new BufferedReader(new FileReader(new
                                    File(name)));
while (file.ready()){ // чтение по условию «пока не конец файла»
str+=file.readLine();}
StringTokenizer str1 = new StringTokenizer(str);
try{  try{
while (str1.hasMoreTokens()){
str = str1.nextToken();
int k =Integer.valueOf(str).intValue();
x.add(new Integer(k));}}
catch(NumberFormatException e){
throw new ErrorData(" ");}
} catch(ErrorData e){
e.Warning();}}
catch(IOException e){
throw new NotFound(" ");}}
catch (NotFound e){
e.Warning();}
return x;}
public static void main(String [] a){
ReadFile R = new ReadFile();
ArrayList array = R.Read();
Massiv m = new Massiv(array);
DataWin a1  = new DataWin(array,"Dimension");
DataWin a2 = new DataWin(m.Metod(),"Repeat elements");}}
```

Порядок выполнения работы

- Получите индивидуальное задание у преподавателя.
- Разработайте структуру классов по индивидуальному заданию.
- Реализуйте по разработанной структуре графическое приложение на языке *Java.*
- сохраните выполненную работу в *jar* архиве.

Лабораторная работа №23. «Разработка и создание программной мини-системы»

Практическое занятие № 1. «Простые сортировки на месте»

Цель работы

Ознакомиться и реализовать простые методы сортировки – сортировки обменом, выбором, вставками, бинарными вставками.

Сортировка обменом

Одним из простых методов сортировки на месте (т.е. при работе не требуется дополнительный объем памяти) является сортировка обменом или «пузырьковая» сортировка.

Суть алгоритма состоит в следующем: сравниваются пары рядом стоящих элементов массива, если первый элемент пары меньше второго, то элементы меняются местами. После первого просмотра массива самый большой элемент встает на свое место, а маленькие по значению элементы на один шаг продвигаются к началу массива. Отсюда метод и получил свое название: «легкие» элементы плавно «всплывают» к началу массива. «Тяжелые» элементы быстро «тонут», встают в конец массива.

После того, как все пары элементов массива просмотрены, массив еще не будет отсортирован, поэтому необходимо просмотреть пары элементов еще раз, и тогда еще один самый большой элемент встанет на свое место.

Если массив состоит из *n* элементов, то пар в таком массиве ровно *n-1*. На каждом шаге алгоритма самый большой элемент массива становится на свое место. Поэтому количество просматриваемых пар уменьшается с каждым шагом на единицу.

Таким образом, в алгоритме явно просматриваются два вложенных цикла. Внутренний цикл отвечает за просмотр пары элементов, количество шагов этого цикла зависит от внешнего цикла. Чем больше шагов выполнил внешний цикл, тем меньше пар просматривает внутренний цикл. Т.к. при выполнении одного шага внешнего цикла самый большой элемент становится на место, то количество шагов цикла равно количеству элементов массива *-1*. Однако, массив может быть уже отсортированным, до окончания работы внешнего цикла. Можно досрочно остановить выполнение цикла, если на каком-то *i*- том шаге во внутреннем цикле не произошло ни одного обмена.

Покажем применение сортировки на массиве *1 5 2 4 3*.

1: 1 2 4 3 ***5***

2: 1 2 3 ***4 5***

3: ***1 2 3 4 5***

После третьего шага алгоритм может закончить свою работу, так как не было проведено ни одного обмена.

Сортировка выбором

Сортировка выбором использует другой принцип упорядочивания элементов. На начальном шаге алгоритма выбирается минимальный элемент и ставится на место нулевого элемента. На последующих, *i*- тых шагах выбирается минимальный элемент среди элементов от *i*- того до (*n-1)*-го.

Рассмотрим на примере массива *1 5 2 4 3.*

1: ***1*** *5 2 4 3* Первый минимальный элемент уже стоит на своем месте.

2: ***1 2*** *5 4 3* Второй минимальный элемент – 2, поменяем его местами с 5.

3: ***1 2 3*** *4 5* Третий минимальный элемент – 3, поменяем его местами с 5.

4: ***1 2 3 4*** *5* Четвертый минимальный элемент стоит на своем месте.

После этого сортировка заканчивается, т.к. последний элемент в любом случае будет стоять на своем месте.

Сортировка вставками

Сортировка вставками упорядочивает массив, выполняя следующие действия – на начальном шаге алгоритма считаем, что последовательность из одного, первого элемента упорядоченная последовательность. Найдем место в этой последовательности для второго элемента. После этого упорядочены уже первые два элемента. Далее в текущей упорядоченной последовательности ищутся места для третьего, четвертого и т.д. элементов.

При программировании поиск места для вставляемого элемента лучше всего осуществить с конца упорядоченной последовательности.

Рассмотрим на примере массива *1 5 2 4 3.*

1: ***1 5*** *2 4 3* В упорядоченную последовательность добавляется 5.

2: ***1 2 5*** *4 3* В упорядоченную последовательность добавляется 2.

3: ***1 2 4 5*** *3* В упорядоченную последовательность добавляется 4.

4: ***1 2 3 4 5*** В упорядоченную последовательность добавляется 3.

Сортировка бинарными вставками

Сортировка бинарными вставками некоторым образом улучшает предыдущий алгоритм, увеличивая скорость нахождения места для вставляемого элемента. Метод использует информацию о том, что часть выборки уже отсортирована. Тогда, можно организовать поиск места для вставки нового элемента следующим образом: пусть переменная F обозначает индекс начала упорядоченной последовательности, а L - индекс конца. Найдем индекс элемента, находящегося в центре отсортированной последовательности - $M = {(F+L)}/{2}$. Сравним вставляемый элемент *X[j]* с элементом *X[M]*. Если *X[j]> X[M]*, то изменим F на M (будем искать в правой части последовательности). Если *X[j]< =X[M]*, то изменим L на M (будем искать в левой части последовательности). Описанные действия будем проводить до тех пор, пока *L>F*. После выхода из цикла место для вставки элемента найдено.

Рассмотрим работу алгоритма на примере уже упорядоченной последовательности:

1 2 3 5 6 7 8 9 10 ***4.***

F=0, L = 8, M = 4 X[4]=6 > 4, следовательно:

F = 0 L = 4 M = 2 X[2]=3 < 4, следовательно:

F = 2 L = 4 M = 3 X[3]=5 >4, следовательно:

F = 2, L = 3 M = 2 (по правилам деления целых чисел в Си 5/2 = 2) X[2]= 3 < 4, следовательно F = 3 L = 3.

Так как F и L равны (условие выполнения цикла), то вставляемый элемент должен занимать в упорядоченной последовательности место с номером 3.

Задание на выполнение

- Получите вариант индивидуального задания у преподавателя.
- Составьте алгоритм простой сортировки.
- Реализуйте алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.
- Выполните полученную программу на случайных массивах размерности 100, 1000, 10000.

Практическое занятие № 2. «Улучшенные сортировки»

Цель работы

Ознакомиться и реализовать один из улучшенных методов сортировок – пирамидальной, Шелла, комбинированной, Хоара. Сравнить время работы простой сортировки и улучшенной.

Пирамидальная сортировка

Пирамида – это частично упорядоченное двоичное дерево, элементы которого расположены в узлах дерева по следующему правилу – каждый элемент родительского узла больше или равен элементов, расположенных в дочерних узлах. Следующий рисунок представляет пирамиду из 15 элементов:

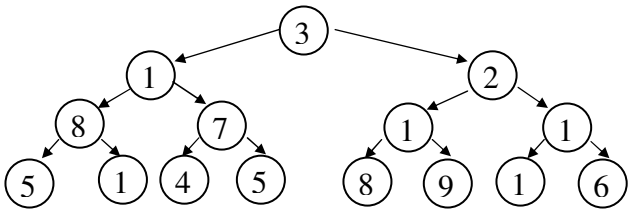


Рис. 2.1. Пирамида из 15 элементов.

Элементы дерева легко представляются в виде массива – пусть родительский узел имеет индекс i, тогда дочерние узлы имеют индексы $2i+1$ и $2i+2$. Рассмотренная пирамида может быть представлена массивом:

Индексы	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Элементы	**30**	**10**	**29**	**8**	**7**	**15**	**11**	**5**	**1**	**4**	**5**	**8**	**9**	**10**	**6**

Т.к. корневой элемент элемент пирамиды всегда является максимальным элементом, то процесс пирамидальной сортировки можно описать следующим образом: поменять верхний элемент пирамиды с нижним элементом и рассматривать в дальнейшем не n элементов исходного массива, а $n-1$ элемент. Но при обмене элементов нарушается правило расположения элементов в пирамиде, поэтому после обмена необходимо перестроить пирамиду с $n-1$ элементами и повторять два этих шага, пока пирамида не останется пустой. Таким образом, необходимо написать процедуру, строящую пирамиду для произвольного массива размерности n, далее алгоритм пирамидальной сортировки очень прост.

Рассмотрим процесс построения пирамиды на произвольном массиве:

Индексы	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Элементы	**25**	**11**	**5**	**11**	**4**	**8**	**3**	**28**	**18**	**10**	**1**	**5**	**4**	**2**	**17**

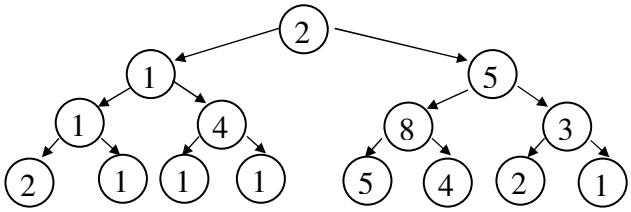


Рис. 2.2. Произвольный массив из 15 элементов.

N = 15. Для элементов, находящихся на нижнем уровне не существует дочерних элементов, т.е. эти элементы могут не проверяться на выполнение правила пирамиды, индексы этих элементов лежат в интервале от *N/2* до *N*. Поэтому построение начинается с элемента с номером *N/2-1*, в примере это *x[6] = 3*, сравним этот элемент с наибольшим из элементов *x[13]* и *x[14]:*

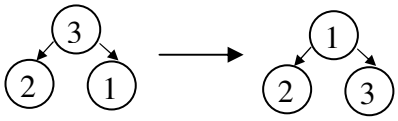


Рис. 2.3. Изменение первой пирамиды
вторая нижняя пирамида остается без изменения, третья и четвертая пирамиды изменяются:

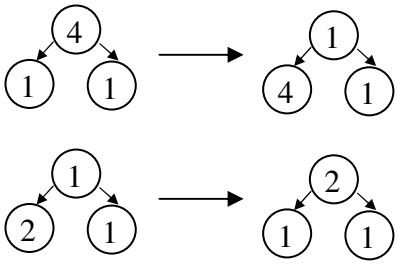


Рис. 2.4. Изменение других пирамид нижнего уровня

Далее рассматриваем пирамиды с узлами во 2-м и 3-м элементах:

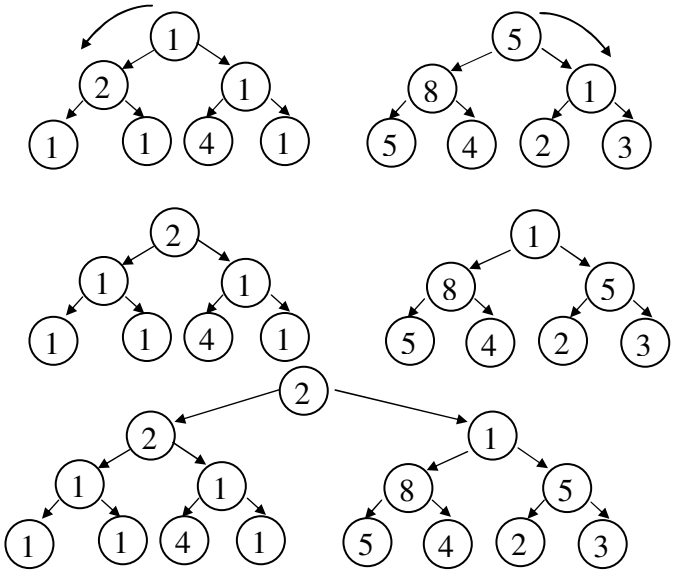


Рис. 2.5. Изменение пирамид второго и первого уровней

Очевидно, что процедура построения пирамиды должна зависеть от номера элемента, на котором строится пирамида, от количества элементов в массиве. Сам массив так же передается параметром.

В общем случае, для построения пирамиды с корнем в *L*-том можно описать следующий алгоритм.

Необходимо осуществить продвижение по дереву вниз:

- элемент с номером *L* меняется местами с большим из своих потомков;
- *L* изменяется на индекс большего потомка;
- алгоритм прекращает работу, когда элемент в позиции *L* больше своих потомков, или когда достигнут нижний уровень, т.е. *L >=N/2-1*.

Сортировка Хоара

Описание классического метода.

Сортировка Хоара использует принцип «разделяй и властвуй». Алгоритм делит весь массив на две части относительно среднего элемента, далее все элементы меньшие среднего элемента переносятся в левый список, а элементы большие среднего элемента в правый список. К двум полученным массивам применяется та же самая процедура. Таким образом, алгоритм является рекурсивным. Процедура сортировки (назовем ее *Qsort*) зависит от границ массива, к которым она применяется *Qsort(nF,nL).*

Первый раз процедура вызывается для *nF = 0, nL = N-1*. Процедура ищет серединное значение массива *(nF+nL)/2*. Значение элемента с таким индексом обозначим *Med = x[(nF+nL)/2]*. Обозначим *i = nF, j = nL.* Пока не найден элемент левого подмассива, больший *Med*, увеличиваем *i*, далее просматриваем правый подмассив, уменьшая *j*, пока не найден элемент, меньший *Med*, если полученное значение *i* меньше или равно *j*, то обменяем найденные значения и перейдем к следующим значениям подмассивов. Увеличение *i* и уменьшение *j* выполняются до тех пор, пока *i<=j*. Далее, если *nF < j*, то вызывается *Qsort(nF,j)*, далее если *nL > i*, то вызывается *Qsort(i,nL).*

Сортировка Хоара с выбором медианного элемента

Можно улучшить быструю сортировку, выбирая средний элемент таким образом, чтобы его значение было бы действительно близким к серединному значению массива.

Для этого можно пользоваться двумя стратегиями:

1. Выбор среднего значения осуществляется случайным образом (с использованием датчиков случайных чисел и информации о размерности массива). Т.к. разделяющий элемент выбирается при каждом вызове процедуры, случайный выбор может быть наиболее правильным и оградит от появления наихудшего случая – когда медианный элемент оказывается наименьшим или наибольшим.
2. Вторая стратегия состоит в случайном выборе 3-х элементов, по одному из начального, конечного и среднего интервалов сортируемого подмассива. Как разделяющий элемент используется среднее из этих трех чисел.

Сортировка Шелла

Сортировку Шелла называют еще сортировкой с уменьшающимся шагом. Метод Шелла сортирует элементы массива, отстоящие друг от друга на заданный интервал H_i. После того, как все элементы массива, отстоящие друг от друга на H_i будут отсортированы, интервал H_i изменяется по правилу $H_{i+1} = (H_i - 1)/2$ (для массивов, содержащих более 500 элемен-

тов) и $H_{i+1} = (H_i - 1)/3$ (для массивов, содержащих менее 500 элементов). За H_0 принимается число элементов массива. Метод заканчивает работу, когда H_i становится меньше 1.

Внутри последовательности можно сортировать любым простым способом.

Пример работы сортировки на массиве:

9 2 3 1 11 12 4 6 9 8 7 10 5 .

$H_0 = 12, H_1 = (12-1)/3 = 3.$ С таким шагом в массиве можно выделить три последовательности:

9 *2 3* ***1*** *11 12* ***4*** *6 9* ***8*** *7 10* ***5*** . После первого шага сортировки массив будет выглядеть следующим образом:

1 *2 3* ***4*** *6 9* ***5*** *7 10* ***8*** *11 12* ***9.*** $H_2 = (3-1)/3 = 0.$ Но массив еще не сортировался с шагом 1, поэтому, следующий и последний шаг сортировки выполняется с шагом 1 :

1 2 3 4 5 6 7 8 9 10 11 12.

Комбинированная сортировка

Комбинация пузырька и сортировки Шелла. На каждом шаге сравниваются значения отстоящие друг от друга на заданное значение шага $H_{i+1} = 8 \cdot H_i / 11$, но такое сравнение происходит всего один раз. Как только значение смещения становится равным 1, выполняется сортировка до конца методом пузырька. За H_0 принимается число элементов массива.

Пример работы сортировки на массиве:

9 2 3 1 11 12 4 6 9 8 7 10 5 .

$H_0 = 12, H_1 = 8 \cdot H_0 / 11 = 8.$ Можно выделить восемь последовательностей, три из них содержат по одному элементу:

$9^1\ 2^2\ 3^3\ 1^4\ 11^5\ 12^6\ 4^7\ 6^8\ \ 9^1\ 8^2\ 7^3\ 10^4\ 5^5$.

После первого шага сортировки массив выглядит следующим образом:

$9^1\ 2^2\ 3^3\ 1^4\ 5^5\ 12^6\ 4^7\ 6^8\ 9^1\ 8^2\ 7^3\ 10^4\ 11^5$. $H_2 = 8 \cdot 8/11 = 5.$

$9^1\ 2^2\ 3^3\ 1^4\ 5^5\ 12^1\ 4^2\ 6^3\ \ 9^4\ 8^5\ 7^1\ 10^2\ 11^3$. После второго шага сортировки:

$9^1\ 2^2\ 3^3\ 1^4\ 5^5\ 7^1\ 4^2\ 6^3\ \ 9^4\ 8^5\ 12^1\ 10^2\ 11^3$. $H_3 = 5 \cdot 8/11 = 3.$

$9^1\ 2^2\ 3^3\ 1^1\ 5^2\ 7^3\ 4^1\ 6^2\ \ 9^3\ 8^1\ 12^2\ 10^3\ 11^1$. После третьего шага сортировки:

$1^1\ 2^2\ 3^3\ 4^1\ 5^2\ 7^3\ 8^1\ 6^2\ \ 9^3\ 9^1\ 12^2\ 10^3\ 11^1$. $H_4 = 3 \cdot 8/11 = 2.$

$1^1\ 2^2\ 3^1\ 4^2\ 5^1\ 7^2\ 8^1\ 6^2\ \ 9^1\ 9^2\ 12^1\ 10^2\ 11^1$. После четвертого шага сортировки:

1 *2* ***3*** *4* ***5*** *6* ***8*** *7* ***9*** *9* ***11*** *10* ***12.***

На пятом шаге сортировки выполняется алгоритм сортировки обменом - *1 2 3 4 5 6 7 8 9 9 10 11 12.* Если при программировании учитывается правило досрочного выхода из внешнего цикла, то алгоритм закончится после выполнения второго внутреннего цикла.

Порядок выполнения работы

- получите вариант задания у преподавателя;
- Составьте алгоритм улучшенной сортировки.
- Реализуйте алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.
- Выполните полученную программу на случайных массивах размерности 100, 1000, 10000.
- Сравните результаты работы простой и улучшенной сортировок по количеству сравнений и перестановок.
- Проанализируйте полученные результаты.

Практическое занятие №3. «Сортировка слиянием»

Цель работы

Ознакомиться и реализовать алгоритм сортировки слиянием.

Сортировка слиянием

Слияние – объединение двух отсортированных подмассивов в один. Таким образом, слияние сортирует два подмассива и сливает их для получения отсортированного массива.

Слияние не зависит от характера входных данных. Основной недостаток метода – необходимость дополнительного объема памяти.

Сортировка слиянием используется в следующих случаях:

- Быстродействие выходит на первое место
- Легко применяется для структур с последовательным доступом к данным
- При решении задач следующего типа – основной набор данных уже отсортирован, в систему постоянно поступают новые данные, которые необходимо разместить, не теряя упорядоченности всего набора данных.

Способы программирования сортировки

Нисходящая сортировка слиянием

Первый способ программирования сортировки очень напоминает способ программирования сортировки Хоара. Функция сортировки рекурсивная, зависит от сортируемого массива и границ сортируемого массива. Запишем алгоритм сортировки:

Используемые обозначения:

X - сортируемый массив

F - индекс начала массива

L – индекс конца массива

Сортировка (X, F, L)

1. ЕСЛИ *L<=F* закончить работу
2. *M = (L+F)/2 //* найти середину
3. *Сортировка(X, F, M) //* вызвать сортировку для левого //подмассива
4. *Сортировка(X,M+1,L)//* вызвать сортировку для правого // подмассива
5. *Слияние (X,F,M,L)*

На рис. 3.1 изображено дерево разбиений, которое строит вышеописанный алгоритм на массиве из 25 элементов (элементы дерева – количество подмассивов).

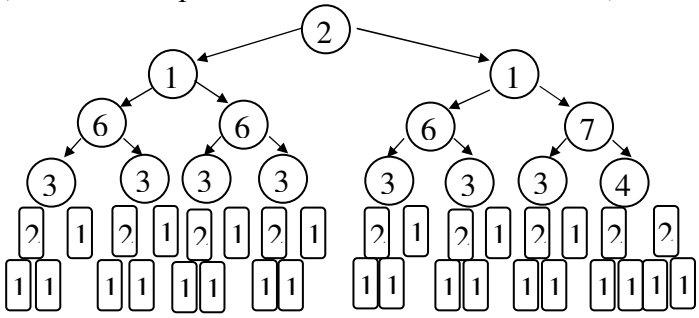


Рис. 3.1. Дерево разбиений, построенное нисходящим слиянием.

Как только алгоритм выполнил разбиение на подмассивы по одному элементу (нижний уровень дерева) начинает свою работу алгоритм слияния.

Восходящая сортировка слиянием

Дерево разбиений, построенное алгоритмом восходящей сортировки представлено на рис. 1.2. Восходящая сортировка слиянием просматривает массив слева направо и на первом шаге сливает рядом стоящие одиночные элементы в упорядоченные пары элементов. На втором шаге – рядом стоящие упорядоченные пары в упорядоченные четверки элементов. И так далее. Алгоритм восходящей сортировки не рекурсивен. Функция, выполняющая слияние вызывается сразу же после выделения границ объединяемых подмассивов.

Слияние подмассивов

Прямое (двухпутевое) слияние.

Алгоритм прямого слияния можно описать следующим образом:
пусть *F* – начало первого подмассива, *L* – конец второго подмассива, *M* – конец первого подмассива, *X* – массив.

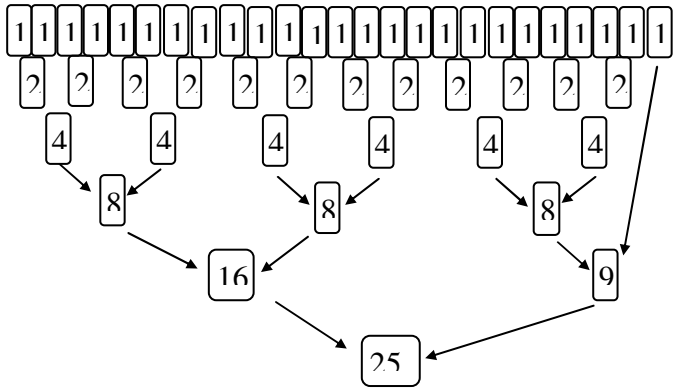


Рис. 3.2. Дерево восходящей сортировки слиянием.

1. *i=F, j=M+1.*
2. ПОКА (*i<=M* И *j<=L*)
   2.1. ЕСЛИ *X[i]<X[j]* ТО записать во вспомогательный массив *X[i].*
   ИНАЧЕ ЕСЛИ *X[j]<X[i]* ТО записать во вспомогательный массив *X[j].*
   ИНАЧЕ записать во вспомогательный массив *X[i]* и *X[j].*
   i++,j++.
   КОНЕЦ ЦИКЛА
3. ЕСЛИ *i<M* ТО переписать остаток первого подмассива во вспомогательный массив.
4. ЕСЛИ *j<L* ТО переписать остаток второго подмассива во вспомогательный массив.
5. Переписать данные из вспомогательного массива в исходный

Абстрактное обменное слияние

При программировании прямого слияния необходимо постоянно отслеживать концы сливаемых подмассивов и случай равенства просматриваемых элементов.

Можно избавиться от этих недостатков, применяя следующую схему слияния:

1. Объединить первый и второй подмассивы во вспомогательный по следующему правилу:
   элементы первого подмассива перепишем в прямом порядке;
   элементы второго подмассива перепишем в обратном порядке;
2. Используем индекс *i* - для первого подмассива (изменяем его слева направо во вспомогательном массиве) и индекс *j* – для просмотра второго подмассива, изменяем его справа налево во вспомогательном массиве.
3. Сравним элементы, находящиеся на позициях *i, j* и переносим в результирующий массив меньший из элементов.

Порядок выполнения

- Получите вариант задания у преподавателя.
- Составьте алгоритм сортировки слиянием.
- Реализуйте алгоритм на языке Си, добавив в программу подсчет количества сравнений и перестановок, проведенных алгоритмом.
- Проанализируйте полученные результаты.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Павловская Т. А. C/C++. Программирование на языке высокого уровня для магистров и бакалавров : учебник для вузов / Т. А. Павловская. - СПб. : ПИТЕР, 2012. - 461 с. (Экземпляры всего: 3).
2. Хабибуллин И. Ш. Программирование на языке высокого уровня C/C++ : учебное пособие для вузов / И. Ш. Хабибуллин. - СПб. : БХВ-Петербург, 2006. - 485[13] с. : (Экземпляры всего: 6).
3. Иванов Б. Н. Дискретная математика. Алгоритмы и программы : Учебное пособие для вузов / Б. Н. Иванов. - М. : Лаборатория Базовых Знаний, 2003. - 288 с. (Экземпляры всего: 50).
4. Новиков Ф. А. Дискретная математика для программистов : Учебное пособие для вузов / Ф. А. Новиков. - 2-е изд. - СПб. ; М. ; Нижний Новгород : Питер, 2007. - 363[5] с. (Экземпляры всего: 80).
5. Вирт Н. Алгоритмы и структуры данных (с примерами на Паскале) : пер. с англ. / Н. Вирт ; пер. Д. Б. Подшивалов. - 2-е изд., испр. . - СПб. : Невский диалект, 2007. - 351[1] с. (Экземпляры всего: 1).

Приложение А

Образец титульного листа к отчету по лабораторной работе

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

Студент гр. 423-1
_____ И.П. Иванов
«___» _____ 20 __ г.

Преподаватель:
ст. преп. каф. АОИ
_____ Н.В. Пермякова
«___» _____ 20 __ г.

2015