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
«___»_____2015 г.

Информатика (Информатика и программирование)

методические указания к лабораторным работам
для студентов направления
080500.62 – «Бизнес-информатика» и
231000.62 – «Программная инженерия»

Разработчик
ст. преподаватель каф. АОИ
_____Н.В. Пермякова
«___»_____2015 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.

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.

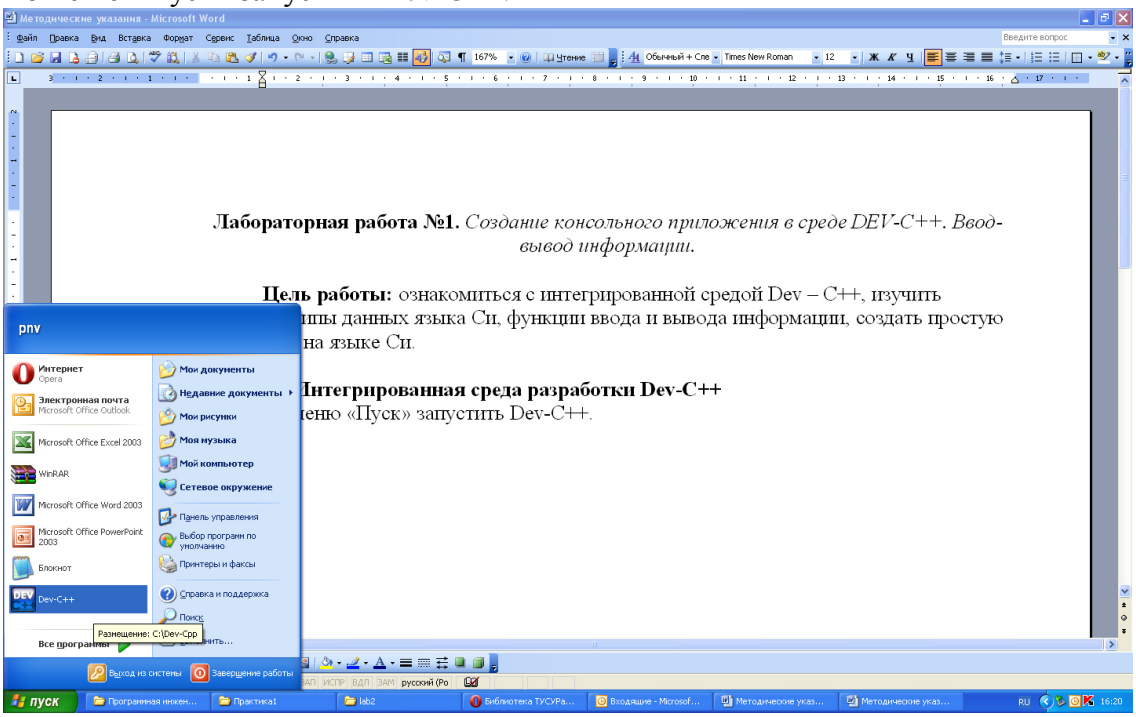
– обучение студентов владению языками структурного программирования, математическим аппаратом систем счисления, навыками использования прикладных программ, навыками разработки и отладки программ на алгоритмических языках программирования.

Лабораторная работа №1. *Создание консольного приложения в среде DEV-C++. Ввод-вывод информации.*

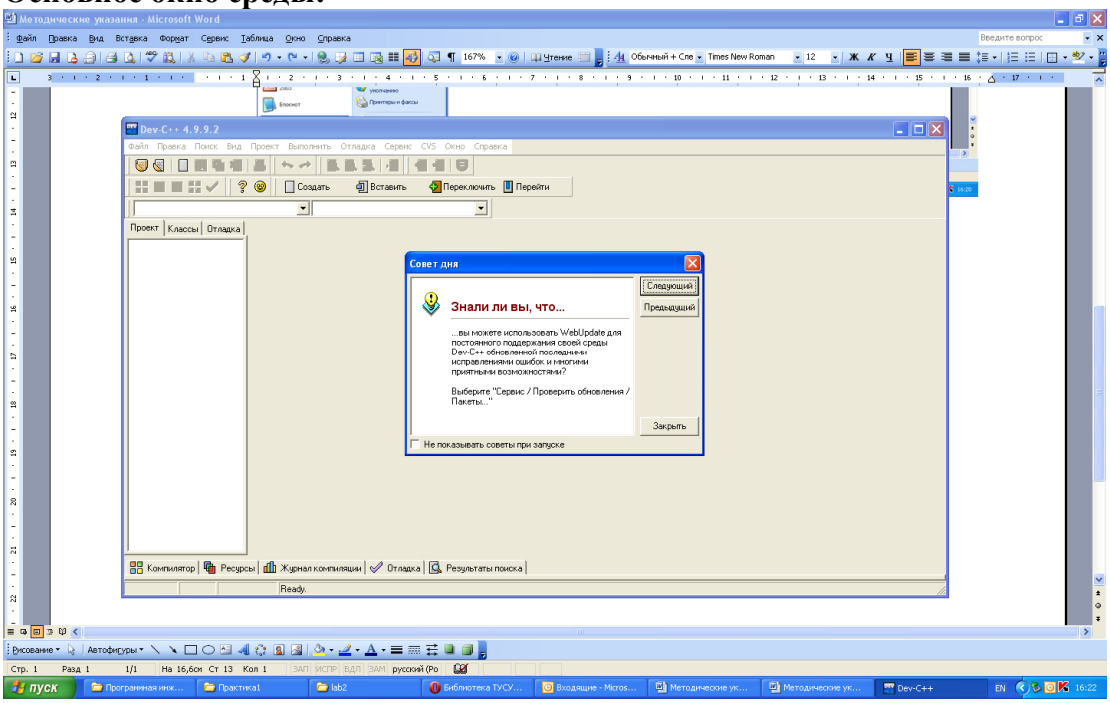
Цель работы: ознакомиться с интегрированной средой Dev – C++, изучить основные типы данных языка Си, функции ввода и вывода информации, создать простую программу на языке Си.

Интегрированная среда разработки Dev-C++

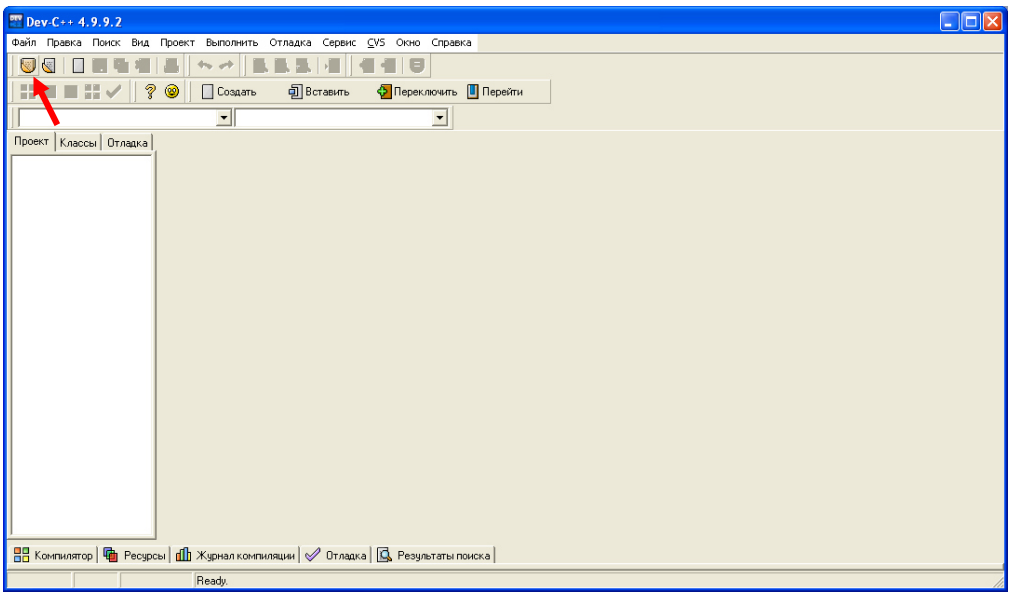
Из меню «Пуск» запустить Dev-C++.



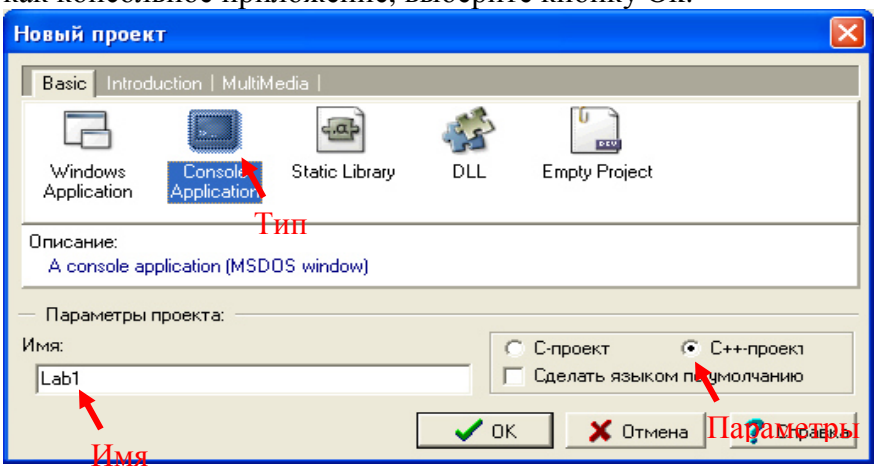
Основное окно среды:



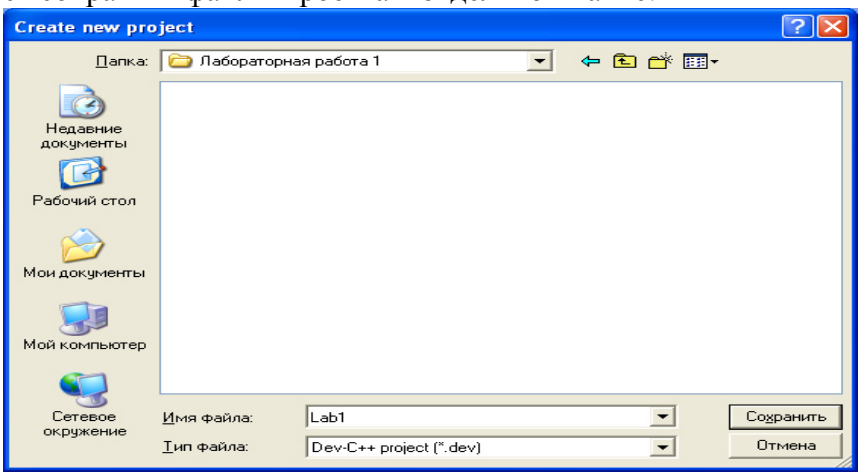
Выберите пункт меню «Создание проекта», либо используя навигацию по системе меню Файл-Создать-Проект, либо воспользовавшись иконкой панели управления:



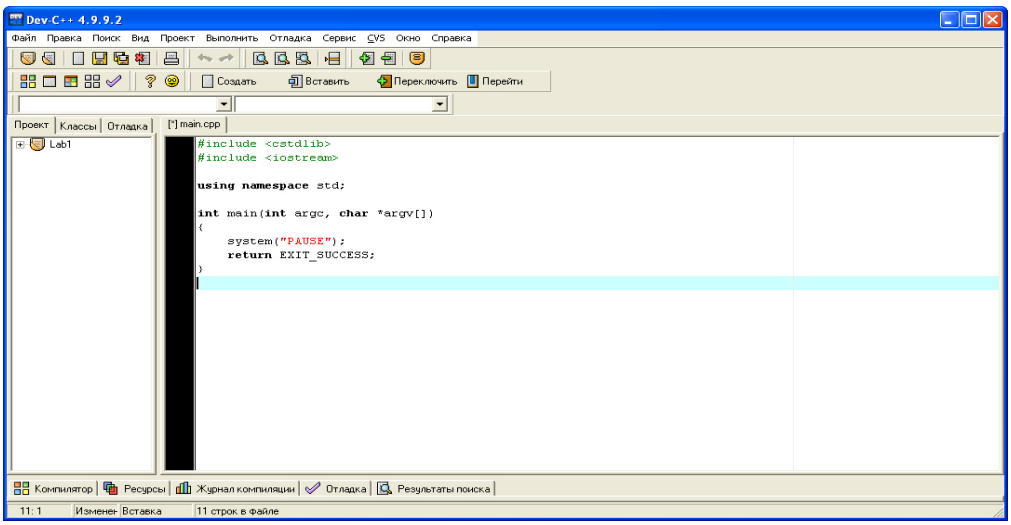
Дайте проекту имя (желательно использовать символы английского алфавита), выберите параметры проекта (Си или Си++), рекомендуется Си++, определите тип проекта как консольное приложение, выберите кнопку Ok:



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 интерпретируется 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);
…
```

Содержание отчета

Отчет по лабораторной работе должен содержать текст индивидуального задания, алгоритм решения задачи, результаты тестирования программы.

Порядок защиты работы

При выполнении задания лабораторной работы в полном объеме студенту может быть выставлено максимально 4 балла рейтинга:

а) 1 балл за компьютерную работу, которая позволяет получать правильное решение задачи;

б) 1 балл за грамотно оформленный отчет;

в) 2 балла за ответы на вопросы как по решению задачи, так и по компьютерной программе.

Порядок выполнения работы

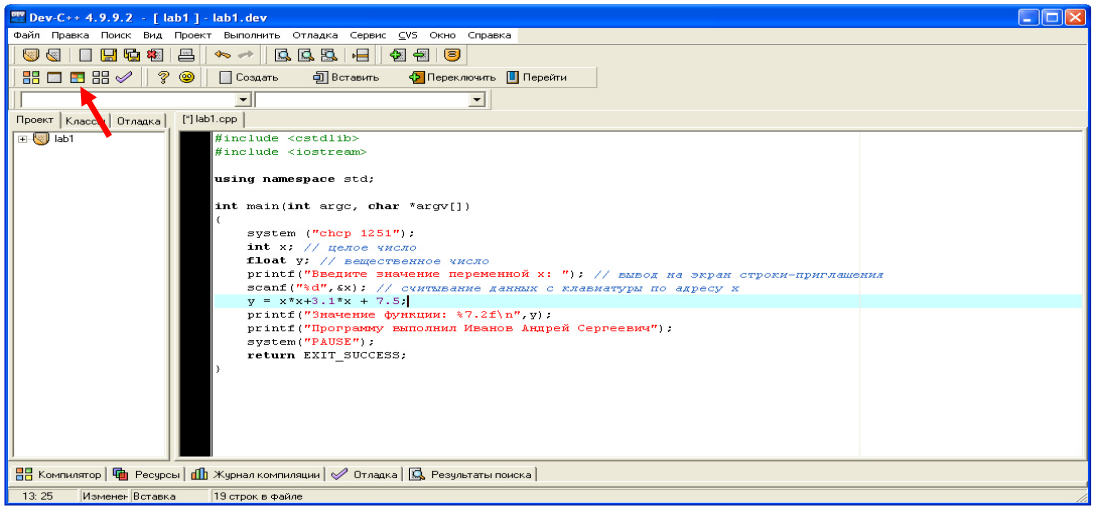
1. Получить индивидуальное задание
2. Создать проект в Dev-C++
3. Описать входные и выходные данные
4. Ввести данные с клавиатуры
5. Вычислить значение функции
6. Вывести полученное значение на экран
7. Вывести личные данные
8. Выполнить компиляцию проекта
9. Оформить отчет

10. Защитить работу

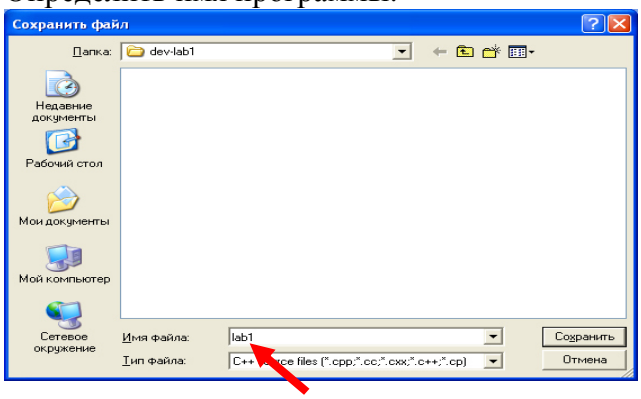
Пример выполнения задания

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

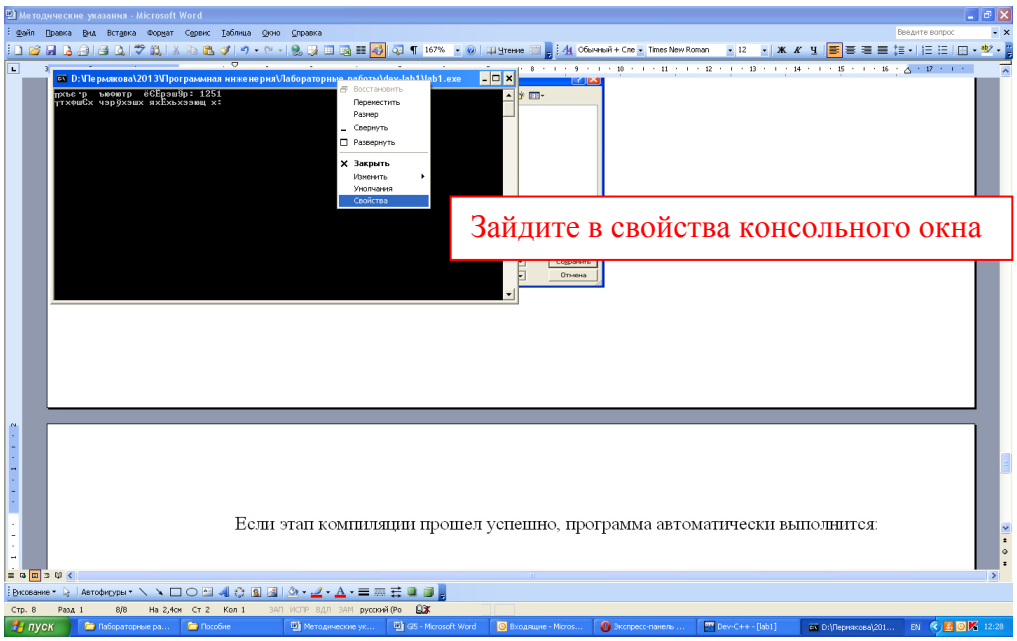
2.	Создание проекта	Запустить Dev – C++, создать новый проект, дополним код программы вызовом функции system ("chcp 1251"); - смена кодировки страницы
3.	Описание переменных	int x; float y;
4.	Ввод данных с клавиатуры	printf(“Введите значение переменной x: ”); scanf(“%d”,&x);
5.	Вычисление значения функции	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”);

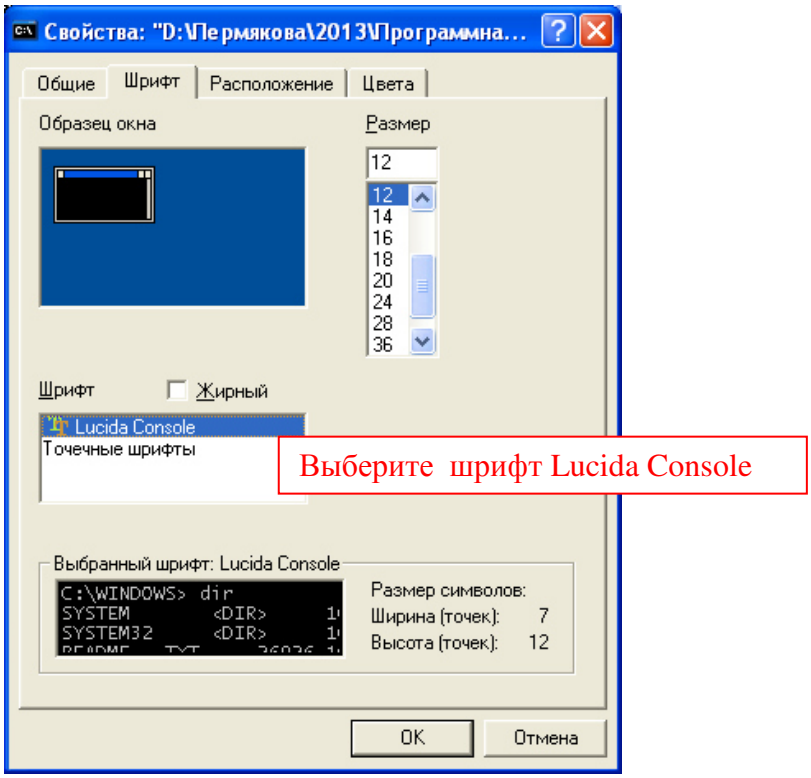


Определить имя программы:

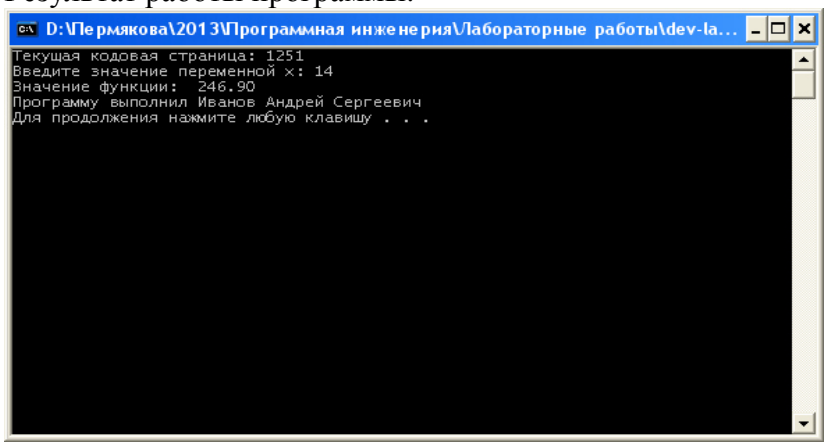


Если этап компиляции прошел успешно, программа автоматически выполнится. Для смены кодировки страницы выполните следующие действия:





Результат работы программы:



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.

Лабораторная работа №2. *Проверка ошибок ввода в языке программирования Си. Проверка условий. Геометрия на плоскости.*

Цель работы: 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.

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).

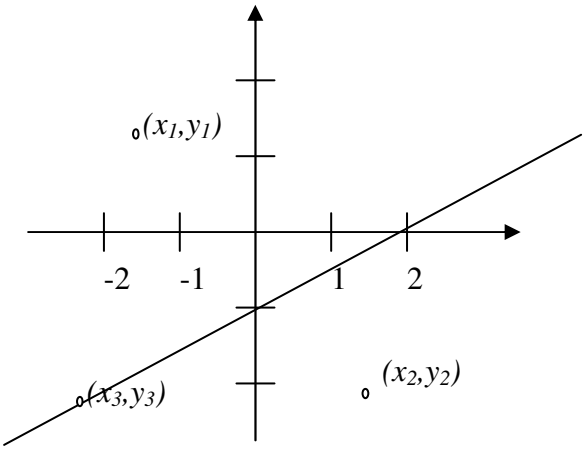


Рис. 2.1. Проверка расположения

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. *).*

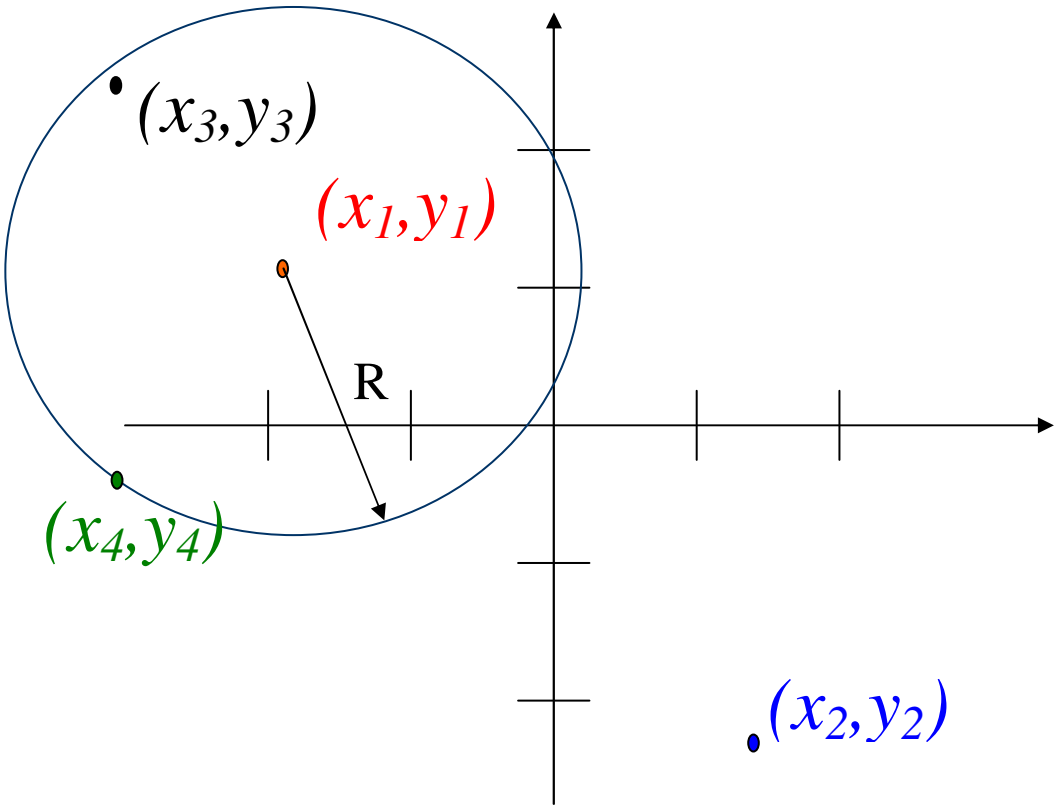


Рис. 2.2. Проверка расположения точки относительно окружности

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).

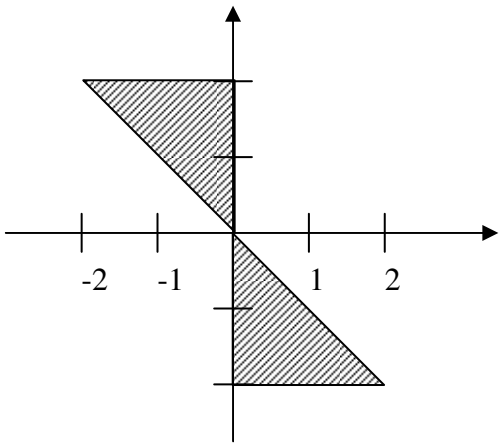


Рис. 2.3. Пример 1.

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:

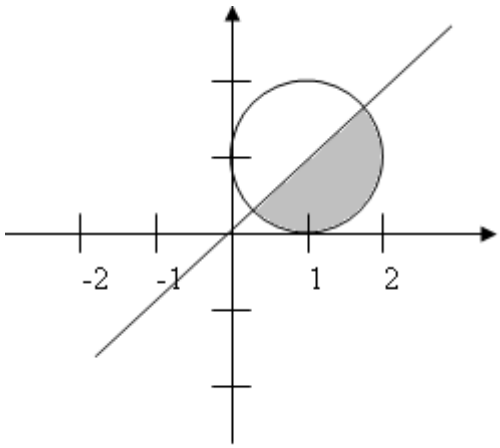


Рис. 2.4. Пример 2.

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. Что возвращает функция *scanf?*
2. Запишите функцию *scanf* для ввода трех переменных.
3. Что Вы понимаете под некорректными данными?
4. Какие данные будут некорректными для решения следующей задачи – “Даны длины трех сторон треугольника. Найдите площадь треугольника.”
5. Как в Си реализована условная конструкция структурного программирования?
6. Опишите синтаксис конструкции if else языка Си.
7. 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?

8. 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?

9. 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;
```

…

10. 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 №3 *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 №4. *Статические массивы в Си*

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

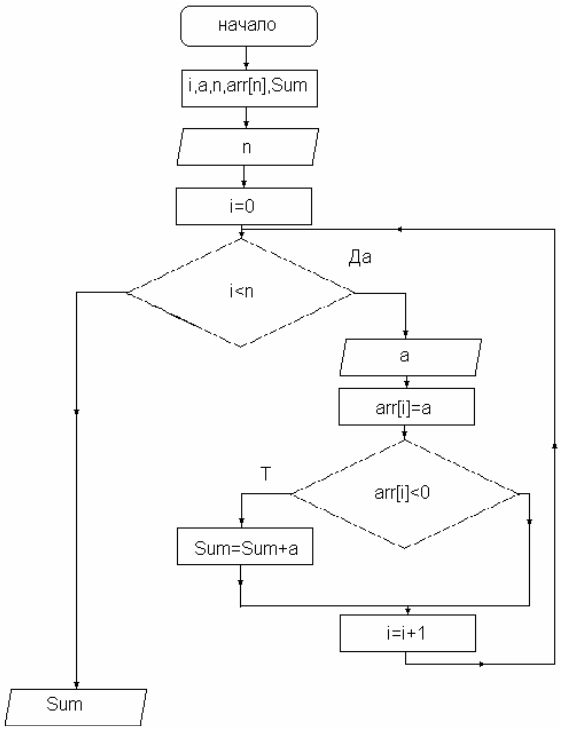
Имя_массива[*номер элемента*]=*значение*;

- 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 №5. *Динамические массивы. Сортировка массивов.*

Цель работы: изучение способов работы с динамическими одномерными массивами; реализация простых алгоритмов сортировки

Динамические массивы в Си.

При решении задач очень часто размер массива является переменной величиной. В этом случае в программах наиболее часто используются динамические массивы. Инициализация таких массивов выполняется в два этапа. Первый этап - выделение памяти, второй этап – инициализация значений элементов массива.

Описание динамического массива:

```
int *x; // описание указателя на целое число
```

Выделение памяти под хранение элементов массива:

```
int n = 50; x=new int[n];// выделение памяти под хранение n элементов
```

Задать элементы массива можно случайным образом:

```
srand(time(NULL)); // рандомизация ядра датчика случайных чисел
//описание функции time() находится в time.h
for(int i=0;i<n;i++){
  x[i] = rand()%30 – rand()%30;// случайное число от в интервале (-30;30)
printf("%4d",x[i]);}
```

Перед окончанием работы с динамическим массивом память, занятая под хранение элементов массива должна быть освобождена:

```
delete [] x;
```

Методы сортировки

Решение некоторых задач требует располагать элементы массива в упорядоченном виде. Процесс расположения элементов в определенном порядке называется сортировкой массива.

Пусть есть последовательность **a_0, a_1... a_n** и **функция сравнения**, которая на любых двух элементах последовательности принимает одно из трех значений: меньше, больше или равно.

Тогда, **задачей сортировки** является перестановка членов последовательности таким образом, чтобы выполнялось условие:

$a_i <= a_{i+1}$, для всех i от 0 до n.

Сортировка обменом

Одним из простых методов сортировки на месте (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 пар 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.

Запишем алгоритм обменной сортировки на псевдокоде.

Задать размерность массива n, сам массив X.
ЦИКЛ ($i = 0; i < n-1; i = i+1$)
    $flag = 0;$ // Обменов нет
    ЦИКЛ ($j = 0; j < n-i-1, j = j+1$)
        ЕСЛИ ($X[j] > X[j+1]$) ТО
                Обмен $X[j] \leftrightarrow X[j+1]$
                $flag = 1;$// Обмен есть
    КОНЕЦ ЦИКЛА
    // Если обменов не было
ЕСЛИ ($flag==0$) ТО
        Закончить выполнение цикла по i.
КОНЕЦ ЦИКЛА
Печать массива X.
Конец алгоритма.

Шейкерная сортировка является улучшением алгоритма сортировки обменом. Если просматривать пары элементов массива с начала, то при каждом проходе на свое место встает самый большой элемент, если же просматривать пары элементов с конца массива, то при каждом просмотре массива на свое место будет вставать самый маленький элемент. Таким

образом, вместо одного внутреннего цикла будем рассматривать массив с начала и до конца и с конца до начала.

Но такое изменение алгоритма не сделает его более «быстрым». Добавим к прямому и обратному проходам массива запоминание места, в котором был произведен последний обмен элементов. Рассмотрим алгоритм шейкерной сортировки на примере следующего массива:

1 0 2 5 8 9 7 4 6 3

Весь массив не отсортирован, поэтому индекс начала F = 0, индекс конца L = n-1 (9)

Прямой проход (от F до L (не включается)):

0 1 2 5 8 7 4 6 3 9 L = 9 (последний обмен 8 и 9 элементов)

Обратный проход: (от L-1 до F)

0 1 2 3 5 8 7 4 6 9 F = 4 (последний обмен 4 и 5 элементов)

Прямой проход (от F до L (не включается)):

0 1 2 3 5 7 4 6 8 9 L = 8

Обратный проход (от L-1 до F)

0 1 2 3 4 5 7 6 8 9 F = 5

Прямой проход (от F до L (не включается)):

0 1 2 3 4 5 6 7 8 9 L = 7

Обратный проход (от L-1 до F)

0 1 2 3 4 5 6 7 8 9 - обменов нет, массив отсортирован

Таким образом, алгоритм тоже имеет два цикла: внешний цикл можно реализовать в виде бесконечного цикла, выход из которого будет осуществляться, как только в одном из внутренних циклов не будет произведено ни одного обмена. Во внешнем цикле выполняются два цикла, которые реализуют прямой и обратный проходы по массиву и обмены элементов, стоящих не на «своих» местах. При этом в прямом проходе изменяется индекс последнего просматриваемого элемента, а в обратном – индекс первого просматриваемого элемента.

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 и ставится на место *i*-того элемента.

Запишем алгоритм сортировки выбором:

Задать размерность массива *n*, сам массив *X*.
ЦИКЛ($i = 0; i < n-1; i = i+1$)
  // Считать минимальным элемент, расположенный на
  //позиции *k*.
  k=i
  ЦИКЛ ($j = i+1; j < n; j = j+1$)
    // Если элемент на позиции *j* меньше минимального,
    // то изменить позицию
    ЕСЛИ $X[j] < X[k]$ ТО *k=j*
  КОНЕЦ ЦИКЛА
  // Если минимальный элемент не стоит на позиции
  // *i*, то выполнить обмен
  ЕСЛИ $k \neq i$ ТО Обмен $X[i] \leftrightarrow X[k]$
КОНЕЦ ЦИКЛА
Печать *X*.
Конец алгоритма

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.

Алгоритм сортировки вставками можно записать следующим образом:

Задать размерность массива *n*, сам массив *X*.
ЦИКЛ($i = 1; i < n; i = i + 1$)
// Сохранить значение вставляемого элемента во
// временной переменной
buf = X[i];
// начать просмотр элементов от *j* до 0
j = i;
// пока текущий элемент меньше предыдущего
ПОКА *buf<X[j-1] И j>0*
    // заменить текущий элемент на предыдущий
    X[j] = X[j-1]
    j = j-1;
КОНЕЦ ЦИКЛА
X[j] = buf;
КОНЕЦ ЦИКЛА
Печать массива
Конец алгоритма

Алгоритм сортировки вставками так же можно улучшить (избавиться от проверки на выход за пределы массива)

Существует 2 способа программирования сортировки вставками со сторожевым элементом:

1 способ.

Перед началом работы в массиве ищется минимальный элемент и выставляется в начало массива. После этого алгоритм работает так же, как описано выше, при этом при реализации цикла ПОКА можно опустить проверку второго условия (*j>0*).

2 способ.

При описании и инициализации массива первая позиция остается пустой, все элементы массива располагаются, начиная с первого (а не с нулевого) элемента. Каждый вставляемый на текущем шаге элемент выставляется в нулевую позицию, тем самым опять же можно опустить второе условие в цикле ПОКА.

Порядок выполнения работы

1. Получить индивидуальный вариант
2. Изучить предложенные методы сортировки
3. Разработать и реализовать на языке Си алгоритм сортировки по индивидуальному варианту
4. Составить отчет по лабораторной работе
5. Защитить работу.

Содержание отчета

1. Индивидуальное задание.

2. Описание алгоритма сортировки.
3. Результаты тестирования программы на массиве из 10 элементов (массив выводить на каждом шаге внутреннего цикла).

Лабораторная работа №6. *Матрицы в языке Си.*

Цель работы: изучение способов работы с динамическими двумерными массивами; реализация простых алгоритмов сортировки и поиска.

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");
 }
```

Порядок выполнения работы

6. Получить индивидуальный вариант
7. Изучить теоретические аспекты лабораторной работы
8. Разработать и реализовать на языке Си алгоритм решения предложенных задач
9. Составить отчет по лабораторной работе
10. Защитить работу.

Содержание отчета

4. Индивидуальное задание.
5. Описание алгоритмов решения задач.
6. Результаты тестирования программы на матрице [5x5].

Лабораторная работа №7. *Функции*

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.

Задание на выполнение

Выполните задание лабораторной работы № 6 с использованием функций. При выполнении задания должны быть написаны функции создания матрицы, печати матрицы, функции, реализующие первую и вторую часть задания. Продемонстрируйте использование написанных функций на двух матрицах разных размерностей.

Порядок выполнения работы

1. Получить индивидуальный вариант задания
2. Написать программу, решающую поставленную задачу с использованием функций
3. Отладить и протестировать программу
4. Написать отчет о проделанной работе
5. Защитить работу

Содержание отчета

1. Задание на лабораторную работу
2. Описание функций (описание функций содержит прототип функции, описание входных и выходных параметров, смысловое описание функции). Например:

   Функция int min(int a1, int a2, int a3)

   // Входные параметры – три целых числа, выходной параметр – целое число.

   // функция возвращает минимальное число из трех заданных *a1, a2, a3*.

Лабораторная работа №8. *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

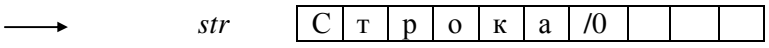
Строка Си может состоять из произвольного количества символов. Окончанием любой строки считается так называемый символ «нуль-терминатор» - '/0'.

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 №9. *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.

9.4.2. 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 №10. *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*
10	*12*	*14*	*16*	*18*
20	*22*	*24*	*26*	*28*
30	*32*	*34*	*36*	*38*
40	*42*	*44*	*46*	*48*

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 №11. *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).

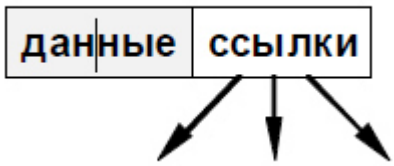


Рис. 11.1. Структура узла списка

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).

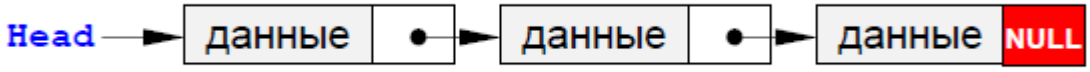


Рис. 11.2. Структура линейного списка

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).

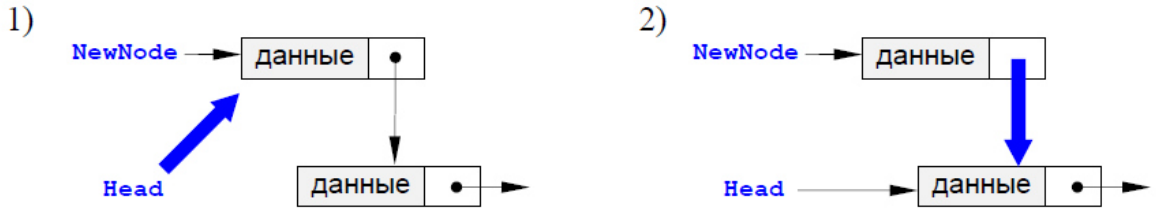


Рис.11.3. Добавление узла в начало списка

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).

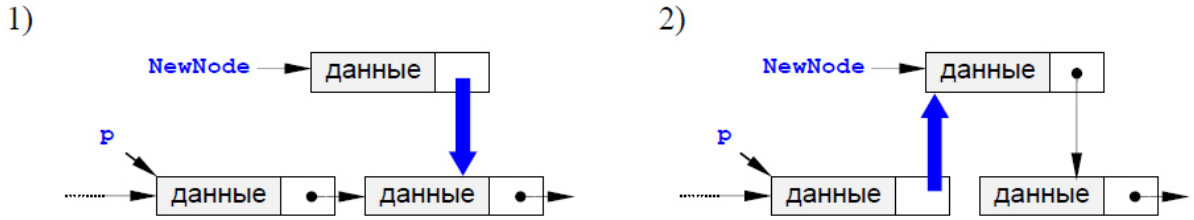


Рис. 11.4. Добавление узла после заданного

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).

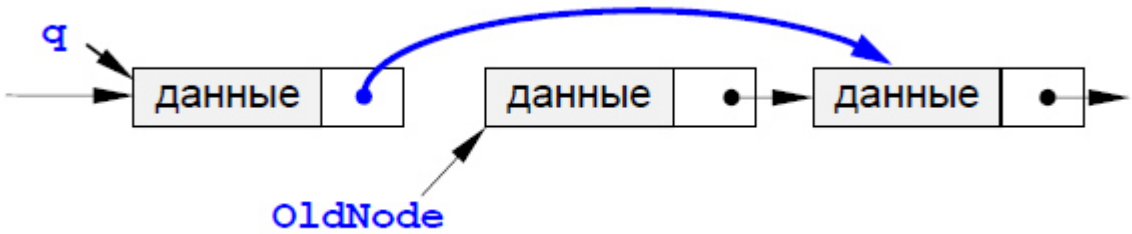


Рис. 11.5. Удаление узла

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).

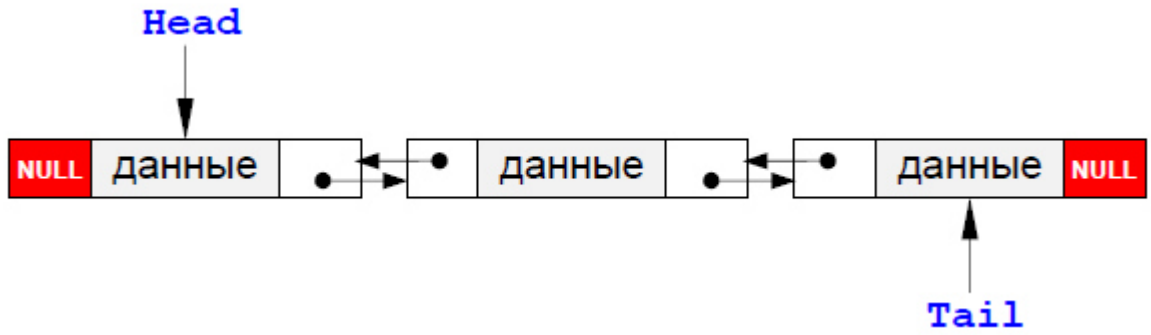


Рис. 11.6. Двунаправленный список

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).

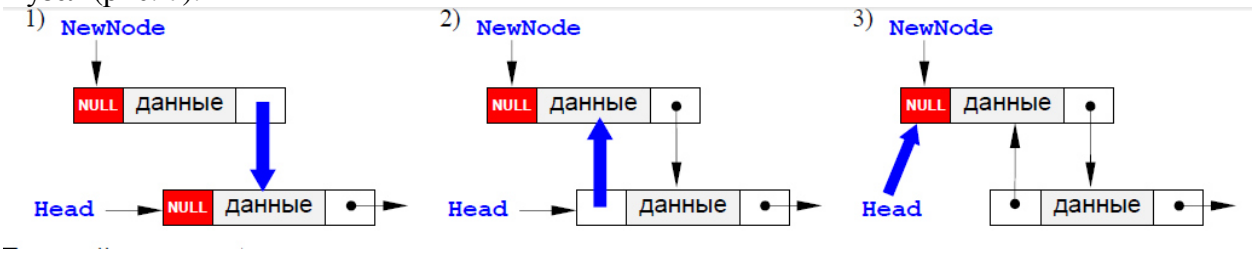


Рис.11.7. Добавление узла в начало списка

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).

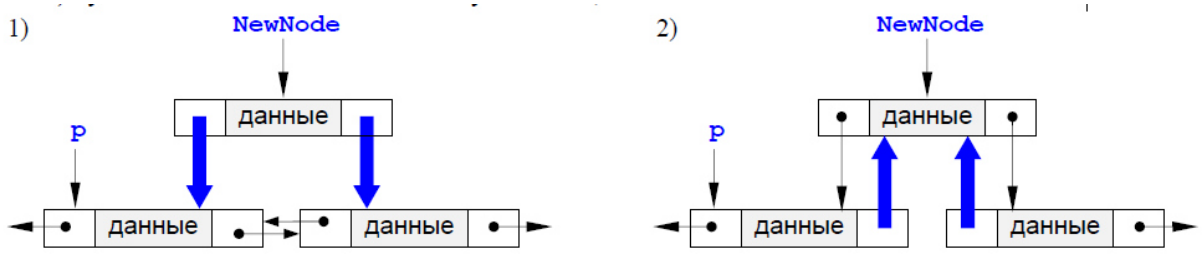


Рис. 11.8. Добавление элемента после заданного

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.

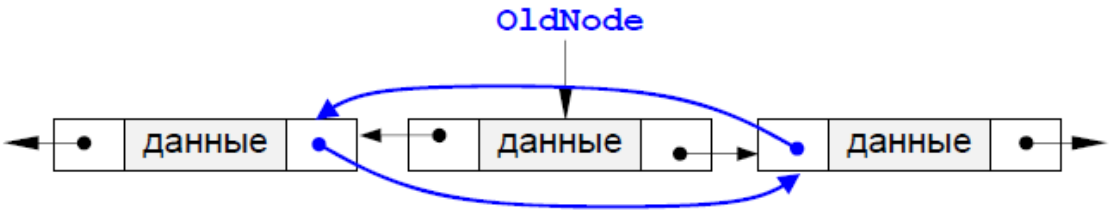


Рис. 11.9. Удаление узла.

```
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 № 12. 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 № 13. *Построение графика функции.*

ОСНОВЫ РИСОВАНИЯ В WIN 32 API
КОНТЕКСТ УСТРОЙСТВА

С точки зрения программиста Windows является системой, не зависящей от устройств (device independent). Эту независимость со стороны Windows обеспечивает библиотека GDI32.dll, а со стороны устройства - драйвер этого устройства.

С точки зрения программы *связующим звеном* между программой и устройством является контекст устройства (Device Context - DC). Если программе нужно осуществить обмен с внешним устройством, программа должна оповестить GDI о необходимости подготовить устройство для операции ввода-вывода. После того, как устройство подготовлено, программа получает дескриптор контекста устройства, т. е. дескриптор структуры, содержащей набор характеристик этого устройства. В этот набор входят:

- bitmap (битовая карта, изображение), отображаемый в окне,
- перо для прорисовки линий,
- кисть,
- палитра,
- шрифт
- и т. д.

После того, как все действия произведены, и необходимость в использовании устройства отпала, программа должна освободить контекст устройства, чтобы не занимать память. В системе может существовать одновременно только ограниченное число контекстов устройств. Если контекст устройства не будет освобождаться после операций вывода, то через несколько перерисовок окна система может зависнуть.

Когда программа требует контекст устройства, она получает его уже заполненным значениями но умолчанию. Программа может создать новый объект, скажем, bitmap или шрифт, и сделать его текущим. Замещенный объект автоматически из памяти не удаляется, его необходимо позже удалить отдельно. Программа может получить характеристики текущего устройства. Изменить эти характеристики можно только через замену объекта.

ТИПЫ КОНТЕКСТА УСТРОЙСТВА

В Windows поддерживаются следующие типы контекстов устройств:

- контекст дисплея (обеспечивает работу с дисплеем);
- контекст принтера (обеспечивает работу с принтером);
- контекст в памяти (моделирует в памяти устройство вывода);
- информационный контекст (служит для получения данных от устройства).

Контекст дисплея

Windows поддерживает три типа контекста дисплея:

- контекст класса,
- приватный контекст
- общий контекст.

Первые два типа используются в приложениях, которые выводят на экран большое количество информации. Например, настольные издательские системы, графические пакеты и т.д.

Приложения, которые не интенсивно работают с экраном, используют общий контекст.

Контекст класса является устаревшим и поддерживается только для обеспечения совместимости с предыдущими версиями Windows.

Контексты устройств хранятся в кэше, управляемом системой. Дескриптор общего контекста программа получает с помощью функций ***GetDC(), GetDCEx()*** или ***BeginPaint().*** После того, как программа отработает с дисплеем, она должна освободить контекст, вызвав функцию ***ReleaseDC()*** или **EndPaint()**. После того, как контекст дисплея освобожден, все изменения, внесенные в него программой, теряются и при повторном получении контекста все действия по изменению контекста необходимо повторять заново.

Приватный контекст отличается от общего тем, что сохраняет изменения даже после того, как прикладная программа освободила его. Приватный контекст хранится в кэше, поэтому прикладная программа может не освобождать его. Естественно, что в этом случае за счет использования большего объема памяти достигается более высокая скорость работы с дисплеем.

Для работы с приватным контекстом необходимо при регистрации класса окна указать стиль CS_OWNDC. После этого программа может получать дескриптор контекста устройства точно так же, как и в случае общего контекста. Система удаляет приватный контекст в том случае, когда удаляется окно.

При работе с контекстами необходимо запомнить, что дескрипторы контекста устройства с помощью функции ***BeginPaint()*** необходимо получать только в случае обработки сообщения **WM PAINT**. Во всех остальных случаях необходимо использовать функции ***GetDC()*** или ***GetDCEx().***

Контекст принтера

При необходимости вывода на принтер программа должна создать контекст устройства с помощью функции **CreateDC().** Аргументы этой функции определяют имя драйвера устройства, тип устройства и инициализационные данные для устройства. Используя эти данные, система может подготовить принтер и распечатать требуемые данные. После распечатки прикладная программа должна удалить контекст принтера с помощью функции ***DeleteDC()*** (не **ReleaseDC**()).

Информационный контекст

Информационный контекст фактически не является контекстом устройства и служит только для получения информации о действительном контексте устройства. Для того, чтобы получить характеристики принтера, программа создает информационный контекст, используя для этого функцию ***CreateIC(),*** а затем из него выбирает требующиеся характеристики. Этот тип контекста создается и работает намного быстрее, а также занимает меньше памяти по сравнению с действительным контекстом. После того, как надобность в информационном контексте миновала, программа должна удалить его с помощью функции **DeleteDC().**

Чаще всего для вывода информации на устройство используется

Контекст в памяти

Этот контекст используется для хранения изображений, которые затем будут скопированы на устройство вывода. Он обязательно создается как совместимый с тем устройством или окном, на которое предполагается копировать информацию Алгоритм работы с контекстом в памяти состоит из нескольких шагов:

1. Получения дескриптора контекста устройства (hDC - handle of Device Context) для окна, в которое будет осуществляться вывод изображения.
2. Получения дескриптора bitmap'a, который будет отображаться в окне.
3. Получения совместимого с hDC контекста в памяти (для хранения изображения) с помощью функции ***CreateCompatibleDC()***
4. Выбора изображения (hBitmap) как текущего для контекста в памяти (hCompatibleDC).
5. Копирования изображения контекста в памяти (hCompatibleDC) на контекст устройства (hDC).

6. Удаления совместимого контекста (hCompatibleDC);

7. Принятия мер для того, чтобы замещенный bitmap из контекста в памяти не остался в памяти.

8. Освобождения контекста устройства (hDC).

Режимы отображения

Подавляющее большинство функций, работающих с оконными координатами, определяют координаты относительно начала рабочей области окна, т. е. от левого верхнего угла.

Таким образом, даже при перемещении окна координаты объектов внутри окна остаются неизменными. При этом единицы, в которых измеряются координаты, зависят от режима отображения (mapping mode), установленного для данного окна. Единицы измерения, зависящие от режима отображения, называют логическими единицами, а координаты в этом случае называют логическими координатами.

При выводе информации на конкретное устройство единицы логических координат преобразуются в физические единицы, которыми являются пиксели.

Для установки текущего режима отображения используется функция ***SetMapping-Mode()***, прототип которой выглядит следующим образом:

WiNGDIAPI int WINAPI SetMapMode(HDC, int); **//** wingdi.h

Первый аргумент - дескриптор контекста устройства, для которого устанавливается данный режим.

Второй аргумент определяет задаваемый режим отображения.

Идентификаторы, использующиеся для обозначения режимов отображения.

Идентификатор	Значение	Эффект
MM_TEXT	1	Логическая единица равна пикселю, начало координат - левый верхний угол окна, положительное значение x - вправо, положительное значение y - вниз (обычный отсчет)
MM_LOMETRIC	2	Логическая единица равна 0, 1 мм, отсчет координат - обычный
MM_HIMETRIC	3	Логическая единица равна 0.01 мм, отсчет координат - обычный
MM_LOENGLISH	4	Логическая единица равна 0,1 дюйма, отсчет координат - обычный
MM_HIENGLISH	5	Логическая единица равна 0,001 дюйма, отсчет координат - обычный

MM_TWIPS	6	Логическая единица равна 1/12 точки на принтере (~ 1/1440 дюйма - «твип»), отсчет координат - обычный
MM_SOTROPIC	7	Логические единицы и направление осей определяются программистом с помощью функций SetWindowExtEx() и SetViewportExtEx(), единицы по осям имеют одинаковый размер
MM_ANISOTROPIC	8	Логические единицы и направления осей определяются так же, как и для MM_ISOTROPIC. но размеры единиц по осям различны
MM_MIN		MM TEXT
MM_MAX		MM_ANISOTROPIC
MM_MAX_FIXEDSCALE		MM_TWIPS

При создании окна по умолчанию устанавливается режим MM_TEXT, т. е. все координаты исчисляются в пикселах.

В папке **Пример1** реализован вывод в окно изображения. Обратите внимание на комментарии в тексте программы.

ПОЛОСЫ ПРОКРУТКИ

Если нет необходимости масштабировать bitmap, но нужно иметь возможность просматривать все части изображения рекомендуется использование полос прокрутки.

В папке **Пример2** реализован вывод изображения в окно с полосами прокрутки.

Для создания окна с полосами прокрутки используются стили - WS__HSCROLL и WS_VSCROLL. Они определяют наличие у окна горизонтальной и вертикальной полос прокрутки соответственно. Управление встроенными полосами прокрутки осуществляется только с помощью курсора мыши.

Для определения диапазона прокрутки (определение числа шагов между крайними позициями слайдера) используется функция ***SetScrollRange(),*** прототип которой в файле ***winuser.h*** определен следующим образом:

WINUSERAPI BOOL WINAPI SetScrollRange(HWND hWnd, int nBar,
int nMinPos, int nMaxPos, BOOL bRedraw);

Первый аргумент функции - дескриптор окна, которому принадлежат полосы прокрутки.

Второй аргумент определяет, для какой полосы прокрутки (вертикальной или горизонтальной) устанавливается диапазон (может принимать значение SB_VERT или SB_HORZ).

Третий и четвертый аргументы непосредственно указывают нижнюю и верхнюю границу диапазона прокрутки.

Пятый аргумент представляет собой флаг, определяющий, нужно ли перерисовывать полосу прокрутки после определения диапазона (TRUE - полоса прокрутки перерисовывается, FALSE - перерисовка не нужна).

Если диапазон прокрутки определен от 0 до 0, то полоса прокрутки становится невидимой.

В примере с помощью функции SetScrollRange() диапазон прокрутки определен как разность между размером bitmap'a и размером окна по вертикали и по горизонтали, т. е. шаг полосы прокрутки соответствует одному пикселю.

Все воздействия на полосу прокрутки приводят к тому, что оконная функция окна, которому принадлежат полосы прокрутки, получает сообщение WM VSCROLL (если действия производились вертикальной полосой) или WM_HSCROLL (реакция на воздействие на горизонтальную полосу).

После того, как зафиксировался факт произведенного с полосой прокрутки действия и характер действия, программа должна правильно отреагировать на него и при необходимости изменить позицию слайдера в соответствии с произведенным воздействием. Это делается с помощью обращения к функции SetScrollPos(), прототип которой выглядит следующим образом:

WINUSERAPI int WINAPI SetScrollPos(HWND hWnd, int nBar. int nPos, BOOL bRedraw);

Первый аргумент - это дескриптор окна, содержащего полосу прокрутки

Второй аргумент может принимать значение SB_VERT или SB_HORZ

Третий аргумент определяет, в какую позицию должен быть установлен слайдер.

Четвертый аргумент определяет, нужно ли перерисовывать полосу прокрутки после установки слайдера.

Для того чтобы в соответствии с новой позицией слайдера изменилось изображение в рабочей области, окну необходимо послать сообщение WM_PAINT, которое заставит окно перерисоваться. Это можно сделать с помощью функции InvalidateRect().

Идентификаторы характеров воздействия на полосы прокрутки

Параметр	Значение	Описание
SB_LINEUP	0	Используется только с WM_VSCROLL; щелчок мышью на стрелке вверх; приводит к прокрутке на одну «строку» вверх .
SB_LUNELEFT	0	Используется только с WM HSCROLL, щелчок мышью на стрелке влево; приводит к прокрутке на одну «колонку» влево
SB_LINEDOWN	1	Используется только с WM_VSCROLL, щелчок мышью на стрелке вниз; приводит к прокрутке на одну «строку» вниз.
SBJJNERIGHT	1	Используется только с WM_HSCROLL, щелчок мышью на стрелке вправо; приводит к прокрутке на одну «колонку» вправо

SB_PAGEUP	2	Используется только с WM_VSCROLL, щелчок мышью на полосе прокрутки выше слайдера; приводит к прокрутке на одну «страницу» вверх
SB_PAGELEFT	2	Используется только с WM_HSCROLL, щелчок мышью на полосе прокрутки левее слайдера; приводит к прокрутке на одну «страницу» влево
SB_PAGEDOWN	3	Используется только с WM_VSCROLL, щелчок мышью на полосе прокрутки ниже слайдера; приводит к прокрутке на одну «страницу» вниз
SB_PAGERIGHT	3	Используется только с WM _HSCROLL, щелчок мышью на полосе прокрутки правее слайдера приводит к прокрутке на одну «страницу» вправо
SB_THUMBPOSITION	4	Перетаскивание слайдера закончено, пользователь отжал клавишу мыши
SB_THUMBTRACK	5	Слайдер перетаскивается с помощью мыши, приводит к перемещению содержимого экрана
SB_ENDSCROLL	8	Пользователь отпустил клавишу мыши после удержания се нажатой на стрелке или на полосе прокрутки

КОНТЕКСТ УСТРОЙСТВА И WM PAINT

В Windows окно само отвечает за перерисовку себя. Для того чтобы окно осуществило перерисовку, оно должно получить сообщение WM_PAINT. Перерисовка окна может быть осуществлена одним из трех методов:

- рабочая область может быть восстановлена, если ее содержимое формируется с помощью каких-либо вычислений;
- последовательность событий, формирующих рабочую область, может быть сохранена, а затем "проиграна" сколь угодно раз (с помощью “метафайлов”);
- можно создать виртуальное окно и направлять весь вывод в виртуальное окно, а при получении основным окном сообщения WM_PAINT копировать содержимое виртуального окна в основное.

В качестве виртуального окна используется контекст в памяти.

РИСОВАНИЕ ГРАФИЧЕСКИХ ПРИМИТИВОВ

Инструменты

Инструментами рисования в Windows являются перо (pen) и кисть (brash). Перо является инструментом для прорисовки линий, цвет и способ заполнения замкнутых графических объектов, таких, как круги, прямоугольники, эллипсы и так называемые регионы, определяются текущей кистью. Во-вторых, рисование невозможно без определения той точки, от которой мы начинаем прорисовку того или иного графического объекта. Обычно эта точка называется текущей графической позицией.

Установка текущей позиции

Для установки текущей позиции используется функция **MoveToEx().** В файле заголовков *wingdi.h* эта функция описывается следующим образом:

WTNGDIAPI BOOL WINAPI MoveToEx(HDC, int, int, LPPOINT);

Первый аргумент - это контекст устройства, на котором мы будем рисовать,

второй и третий - координаты точки, в которую мы устанавливаем текущую графическую позицию.

Последний аргумент - указатель на структуру типа **POINT**, в которую функция запишет координаты старой текущей позиции.

Структура типа ***POINT*** описана в файле **windef.h** и ее описание выглядит следующим образом:

typedef struct tagPOINT
LONG x; LONG y; }
POINT, *PPOINT,
NEAR *NPPOINT,
FAR *LPPOINT;

Если при вызове функции указатель на структуру типа **POINT** равен **NULL**, то координаты старой текущей позиции не возвращаются.

Прорисовка одного пикселя

Прорисовать один пиксель в определенной позиции мы можем с помощью вызова функции ***SetPixel()***, описанной в *wingdi.h*:

WINGDIAPF COLORREF WINAPI SetPixel(HDC, int, int, COLORREF);

Первые три аргумента - контекст устройства вывода и координаты прорисовываемого пикселя.

Создание пера для рисования линий

Рисование графических примитивов производится с помощью перьев. В Windows'95 есть три предопределенных пера –

- черное (BLACK_PEN),
- белое (WHITE_PEN) и
- прозрачное (NULL_PEN).

При создании окна по умолчанию ему присваивается черное перо. Дескриптор каждого из них может быть получен с помощью функции ***GetStockObject().*** Для прорисовки линий можно воспользоваться пером, созданным в программе посредством вызова функции ***CreatePen()***. Ее описание выглядит следующим образом:

WINGDIAPI HPEN WINAPI CreatePen(int, int, COLORREF);

Первый аргумент определяет стиль кисти.

Второй аргумент - толщина пера в логических единицах. Если этот аргумент равен 0, то толщина пера делается равной одному пикселю.

Третий аргумент - цвет чернил.

Теперь для того, чтобы мы могли использовать наше перо, необходимо сделать его текущим в контексте устройства. Делается это уже давно знакомой нам функцией ***SelectObject().*** После работы с пером, необходимо удалить его, вызвав функцию ***DeleteObject().***

Рисование линии

Нарисовать линию можно с помощью функции ***LineTo().***

Ее прототип:

WINGDIAPI BOOL WINAPI LineTo(HDC, int, int);

Первый аргумент - контекст устройства.

Второй и третий аргументы -координаты точки, *ДО КОТОРОЙ ОТ ТЕКУЩЕЙ ПОЗИЦИИ* будет проведена линия. При успешном завершении функция возвращает TRUE.

Текущая позиция будет находиться там, где закончилась линия.

Рисование прямоугольника

Прямоугольник можно нарисовать, обратившись к функции ***Rectangle().***

Её прототип:

WINGDIAPI BOOL WINAPI Rectangle(HDC, int, int, int, int);

Первый аргумент - дескриптор контекста устройства.

Остальные аргументы - координаты верхнего левого и нижнего правого углов прямоугольника. TRUE возвращается при нормальном завершении операции. Прямоугольник автоматически заполняется цветом и способом, определяемым текущей кистью.

Рисование эллипса

Для рисования эллипса необходимо вызвать функцию Ellipse(),

WINGDIAPI BOOL WINAPI Ellipse(HDC, int, int, int, int);

Первый аргумент - контекст устройства.

Второй и третий аргументы - координаты левого верхнего угла прямоугольника, в который вписан эллипс; четвертый и пятый аргументы - координаты нижнего правого угла.

Окружность является частным случаем эллипса.

Как эллипс, так и окружность после прорисовки заполняются цветом и атрибутами текущей кисти.

Рисование прямоугольника с закругленными краями

Прямоугольник с закругленными краями рисуется с помощью функции RoundRect().

WINGDIAPI BOOL WINAPI RoundRect(HDC, int, int, int, int. int. int);

Первые пять аргументов полностью идентичны аргументам функции **Rect().** Последние два аргумента содержат ширину и высоту эллипса, определяющего дуги. После прорисовки прямоугольник закрашивается текущей кистью. В случае успешного завершения функция возвращает TRUE.

Рисование дуги и сектора эллипса

WINGDIAPI BOOL WINAPI Arc(HDC, int, int, int, int, inl, int. int, int);

Первые пять аргументов полностью аналогичны аргументам функции **Ellipse**(). Непосредственно дуга определяется ещё двумя точками. Первая - начало дуги - находится на пересечении эллипса, частью которого является дуга, и прямой, проходящей через центр прямоугольника и точку начала дуги. Вторая - конец дуги - определяется аналогично. Дуга прорисовывается против часовой стрелки.

У функции **Pie(),** которая применяется для рисования сектора эллипса, набор аргументов и их назначение абсолютно идентичны функции ***Arc().***

Заполнение объектов

Заполнение замкнутых графических объектов происходит с помощью текущей кисти. Программист может использовать предопределенную кисть, а может создать свою собственную, после чего сделать ее текущей с помощью функции ***SelectObject().***

Простейшим видом кисти является так называемая сплошная кисть, которая создается с помощью функции ***CreateSolidBrush():***

WINGDIAPI HBRUSH WINAPI CreateSolidBrush(COLOREF)

Единственный аргумент этой функции - цвет кисти

Штриховая кисть

WINGDIAPI HBRUSH WINAPI CreateHatchBrush(int, COLORREF);

Первый аргумент этой функции - стиль штриховки.

Второй аргумент указывает цвет штриховки.

WINGDIAPI HBRUSH WINAPI CreatePatternBrush(HBITMAP);
Единственным аргументом этой функции является дескриптор bitmap'a.
Эти функции при успешном завершении возвращают дескриптор созданной кисти.

РИСОВАНИЕ ГРАФИКА ФУНКЦИИ

1. Рисование осей координат
2. Определение оконных координат точки пересечения осей координат (x0,y0)
3. Масштабирование
   Определение max значения по оси X
   Определение max значения по оси Y
   Определение размера единичного отрезка по осям X и Y(dx, dy)
4. Вычисление текущих значений x и f(x)
   curx = 0;
   cury = f(curx);
5. Вычисление оконных координат

x = x0+curx*dx;
y = y0 + cury*dy;

6. Установить текущую позицию в точку с координатами (x,y)
7. Пока (curx<Mx)
   Изменить curx;
   вычислить cury;
   пересчитать экранные координаты (x,y)
   нарисовать линию
8. **Конец**

УКАЗАТЕЛИ И ФУНКЦИИ

В Си можно создать указатель на функцию. На самом деле имя функции, это указатель-константа, равный адресу первой машинной команды функции. Объявления:

Возвращаемый тип (*имя переменной) (список аргументов)

Используется для передачи функции как параметра в другие функции, для написания рези-

дентных программ, используются в некоторых библиотечных функциях, как параметры.

Например:

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

float ff1(float x)

{
 return x+1;
}
float ff2(float x)

{
 return 2*x;
}
void print_f(float (*funct)(float))
```

```
{
float x;
printf("*****************************************\n");
printf("********  Значения функции на [0;10] *******\n");
printf("*****************************************\n");

for (x=0;x<=10;x++)
        printf("**x = %5.3f ********* f(x) = %5.3f ********\n",x,funct(x));
printf("\n");
}
void main()

{

float (*f_ptr)(float);

clrscr();
f_ptr = ff1;
print_f(f_ptr);
f_ptr = ff2;
print_f(f_ptr);
getch();
}
```

Порядок выполнения работы

1. Написать программу для построения графиков трех заданных функций.
2. Оформить в виде функций:
   - Рисование графика;
   - Вычисление значения заданной функции f(x).
3. Использовать передачу функции f (x) параметром в функцию рисования графика.
4. Типы функций в программе –
   - Степенная (например: $f(x) = 3x^4 - 4x^3 - 12x^2 + 2$)
   - Гармоническая (например: $f(x) = \cos(x) + 5x^2 - 2x$)
   - Разрывная (например: $f(x) = \frac{1 - x^3}{x} - 7x$)

Правила оценивания задания.

1. Рисование графика – 2 балла.
2. Реализация интерфейса – 2 балла.
3. Разметка графика – 2 балла.
4. Передача функции параметром– 2 балла.
5. Возможность изменения масштаба– 2 балла.

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