Кафедра: Автоматика и Вычислительная Техника

Модульное программирование

Содержание

1. Классы памяти идентификатора

1.1 Область действия

1.2 Область видимости

1.3 Продолжительность жизни

2. Модели памяти ОЗУ

2.1 Виды моделей памяти

2.2 Размещение исполняемого файла в ОЗУ в модели large

3. Передача данных в функцию

3.1 Передача параметров по значению

3.2 Передача параметров по адресу

3.3 Передача одномерных массивов

3.4 Передача двумерных массивов

4. Тестирование функций

5. Практические задания

5.1 Указать классы памяти переменной

5.2 Работаем с адресами

5.3 Прототипы функций

5.3.1 Свопинг

5.3.2 Индексы максимальных элементов одномерного массива

5.3.3 Индексы максимальных элементов двумерного массива

5.3.4 Угол между двумя векторами

5.3.5 Определитель матрицы

5.4 Выделение фрагмента программы в отдельную функцию

5.5 Тестирование функции

5.5.1 Сортировка массива

5.5.2 МиниМакс

6. Лабораторные задания

6.1 Линейное уравнение

6.2 Четные элементы массива

6.3 Нахождение простых чисел

6.4 Количество вхождений подстроки в строку

6.5 Произведение матриц

7. Дополнительные задания

Библиографический список

## 1. Классы памяти идентификатора

С каждым объектом памяти переменной связаны следующие понятия: область действия, область видимости и продолжительность жизни. Эти три понятия объединяются в понятие класс памяти переменной. Класс памяти устанавливается синтаксисом и местом размещения определения этой переменной.

Спецификатор класса памяти в объявлении переменной может быть auto, register, static или extern. Если класс памяти не указан, то он определяется по умолчанию из контекста объявления.

Объекты классов auto и register имеют локальное время жизни. Спецификаторы static и extern определяют объекты с глобальным временем жизни.

При объявлении переменной на внутреннем уровне может быть использован любой из четырех спецификаторов класса памяти, а если он не указан, то подразумевается класс памяти auto.

Переменная с классом памяти auto имеет локальное время жизни и видна только в блоке, в котором объявлена. Память для такой переменной выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть выделен другой участок памяти.

Переменная с классом памяти auto автоматически не инициализируется. Она должна быть проинициализирована явно при объявлении путем присвоения ей начального значения. Значение неинициализированной переменной с классом памяти auto считается неопределенным.

Спецификатор класса памяти register предписывает компилятору распределить память для переменной в регистре, если это представляется возможным. Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной. Переменная, объявленная с классом памяти register, имеет ту же область видимости, что и переменная auto. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в том случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память как для класса auto. Класс памяти register может быть указан только для переменных с типом int или указателей с размером, равным размеру int.

Переменные, объявленные на внутреннем уровне со спецификатором класса памяти static, обеспечиваю возможность сохранить значение переменной при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена. В отличие от переменных с классом auto, память для которых выделяется в стеке, для переменных с классом static память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока.

Пример:

/\* объявления переменной i на внутреннем уровне с классом памяти static. \*/

/\* исходный файл file1. c \*/

main ()

{...

}

fun1 ()

{ static int i=0;...

}

/\* исходный файл file2. c \*/

fun2 ()

{ static int i=0;...

}

fun3 ()

{ static int i=0;...

}

В приведенном примере объявлены три разные переменные с классом памяти static, имеющие одинаковые имена i. Каждая из этих переменных имеет глобальное время жизни, но видима только в том блоке (функции), в которой она объявлена. Эти переменные можно использовать для подсчета числа обращений к каждой из трех функций.

Переменные класса памяти static могут быть инициализированы константным выражением. Если явной инициализации нет, то такой переменной присваивается нулевое значение. При инициализации константным адресным выражением можно использовать адреса любых внешних объектов, кроме адресов объектов с классом памяти auto, так как адрес последних не является константой и изменяется при каждом входе в блок. Инициализация выполняется один раз при первом входе в блок.

Переменная, объявленная локально с классом памяти extern, является ссылкой на переменную с тем же самым именем, определенную глобально в одном из исходных файлов программы. Цель такого объявления состоит в том, чтобы сделать определение переменной глобального уровня видимым внутри блока.

Пример:

/\* объявления переменной i, являющейся именем внешнего массива длинных целых чисел, на локальном уровне\*/

/\* исходный файл file1. c \*/

main ()

{...

}

fun1 ()

{ extern long i [] ;...

}

/\* исходный файл file2. c \*/

long i [MAX] ={0};

fun2 ()

{...

}

fun3 ()

{...

}

Объявление переменной i [] как extern в приведенном примере делает ее видимой внутри функции fun1. Определение этой переменной находится в файле file2. c на глобальном уровне и должно быть только одно, в то время как объявлений с классом памяти extern может быть несколько.

Объявление с классом памяти extern требуется при необходимости использовать переменную, описанную в текущем исходном файле, но ниже по тексту программы, т.е. до выполнения ее глобального определения. Следующий пример иллюстрирует такое использование переменной с именем st.

Пример:

main ()

{ extern int st [] ;...

}

static int st [MAX] ={0};

fun1 ()

{...

}

Объявление переменной со спецификатором extern информирует компилятор о том, что память для переменной выделять не требуется, так как это выполнено где-то в другом месте программы.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти static или extern, а так же можно объявлять переменные без указания класса памяти. Классы памяти auto и register для глобального объявления недопустимы.

Объявление переменных на глобальном уровне - это или определение переменных, или ссылки на определения, сделанные в другом месте программы. Объявление глобальной переменной, которое инициализирует эту переменную (явно или неявно), является определением переменной. Определение на глобальном уровне может задаваться в следующих формах:

1. Переменная объявлена с классом памяти static. Такая переменная может быть инициализирована явно константным выражением, или по умолчанию нулевым значением. То есть обьявления static int i=0 и static int i эквивалентны, и в обоих случаях переменной i будет присвоено значение 0.

2. Переменная объявлена без указания класса памяти, но с явной инициализацией. Такой переменной по умолчанию присваивается класс памяти static. То есть объявления int i=1 и static int i=1 будут эквивалентны.

Переменная объявленная глобально видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима (если только она не объявлена с классом extern).

Глобальная переменная может быть определена только один раз в пределах своей области видимости. В другом исходном файле может быть объявлена другая глобальная переменная с таким же именем и с классом памяти static, конфликта при этом не возникает, так как каждая из этих переменных будет видимой только в своем исходном файле.

Спецификатор класса памяти extern для глобальных переменных используется, как и для локального объявления, в качестве ссылки на переменную, объявленную в другом месте программы, т.е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором сделано объявление.

В объявлениях с классом памяти extern не допускается инициализация, так как эти объявления ссылаются на уже существующие и определенные ранее переменные.

Переменная, на которую делается ссылка с помощью спецификатора extern, может быть определена только один раз в одном из исходных файлов программы.

## 1.1 Область действия

Область действия переменной - это часть программы или исходного модуля, в которой определен объект соответствующий этой переменной. Перечислим виды областей действия.

Блок. Эта область действия начинается в точке определения переменной и заканчивается в конце блока, содержащего это определение. Такой блок называется окружающим. Тело функции рассматривается как блок.

Функция. Фактически совпадает с телом функции. Такой областью действия обладают формальные аргументы, перечисленные в заголовке функции, а также метки, используемые операторами goto.

Файл. Распространятся от точки определения переменной до конца файла. Переменные с такой областью действия называются глобальными и должны определяться вне всякой функции. Если глобальная переменная определена в конце файла, то ее область действия фактически пустая и вверх на весь файл не распространяется. Прототип. Такой областью действия обладают переменные из списка формальных параметров прототипа функции.

## 1.2 Область видимости

Видимость переменных и функций в программе определяется следующими правилами.

1. Переменная, объявленная или определенная глобально, видима от точки объявления или определения до конца исходного файла. Можно сделать переменную видимой и в других исходных файлах, для чего в этих файлах следует ее объявить с классом памяти extern.

2. Переменная, объявленная или определенная локально, видима от точки объявления или определения до конца текущего блока. Такая переменная называется локальной.

3. Переменные из объемлющих блоков, включая переменные объявленные на глобальном уровне, видимы во внутренних блоках. Эту видимость называют вложенной. Если переменная, объявленная внутри блока, имеет то же имя, что и переменная, объявленная в объемлющем блоке, то это разные переменные, и переменная из объемлющего блока во внутреннем блоке будет невидимой.

4. Функции с классом памяти static видимы только в исходном файле, в котором они определены.

Метки в функциях видимы на протяжении всей функции.

Имена формальных параметров, объявленные в списке параметров прототипа функции, видимы только от точки объявления параметра до конца объявления функции.

## 1.3 Продолжительность жизни

Время жизни переменной (глобальной или локальной) определяется по следующим правилам.

1. Переменная, объявленная глобально (т.е. вне всех блоков), существует на протяжении всего времени выполнения программы.

2. Локальные переменные (т.е. объявленные внутри блока) с классом памяти register или auto, имеют время жизни только на период выполнения того блока, в котором они объявлены. Если локальная переменная объявлена с классом памяти static или extern, то она имеет время жизни на период выполнения всей программы.

## 2. Модели памяти ОЗУ

Память микропроцессора Intel 80x86 имеет сегментированную архитектуру. Непосредственно можно адресоваться к 64К памяти сегменту. Процессор 80x86 отслеживает 4 различных сегмента: сегмент кода, сегмент данных, сегмент стека и дополнительный сегмент. Кодовый сегмент содержит машинные команды программы; в сегменте данных хранится информация; сегмент стека имеет организацию и назначение стека; вспомогательный сегмент используется для хранения некоторых вспомогательных данных. Процессор 80x86 имеет четыре 16-разрядных сегментных регистра (по одному на сегмент) - CS, DS, SS и ES, которые указывают на сегмент кода, данных, стека и дополнительный сегмент соответственно. Сегмент может находиться в любом месте памяти, но начинаться должен по адресу, кратному 10. Сегменты могут перекрываться. Например, все четыре сегмента могут начинаться с одного адреса.

Полный адрес в 8086 состоит из двух 16-битовых значений: адреса сегмента и смещения. Предположим, что адрес сегмента данных - т.е. значение в регистре DS - равен 2F84 (шестнадцатеричное) и вы желаете вычислить фактический адрес некоторого элемента данных, который имеет значение 0532 (основание 16) от начала сегмента данных; как это сделать?

Вычисление адреса будет выполнено следующим образом: нужно сдвинуть влево значение сегментного регистра на 4 бита (это эквивалентно одной шестнадцатеричной цифре), а затем сложить с величиной смещения.

Полученное 20-битовое значение и есть фактический адрес данных, как показано ниже:

регистр DS (после сдвига): 0010 1111 1000 0100 0000 = 2F840

смещение: 0000 0101 0011 0010 = 00532

---------------------- - -------------------------------

Адрес: 0010 1111 1101 0111 0010 = 2FD72

Участок памяти величиной 16 байт называется параграфом, поэтому говорят, что сегмент всегда начинается на границе параграфа.

Начальный адрес сегмента всегда является 20-битовым числом, но сегментный регистр имеет всего 16 битов - поэтому младшие 4 бита всегда предполагаются равными нулю. Это означает - как было уже сказано - что начало сегмента может находиться только в адресах памяти, кратных 16, т.е. адресах, в которых последние 4 бита (или один шестнадцатиричный разряд) равен нулю. Поэтому если регистр DS содержит значение 2F84, то фактически сегмент данных начинается в адресе 2F840.

Стандартная запись адреса имеет форму *сегмент: смещение*; например, предыдущий адрес можно записать как 2F84: 0532. Отметим, что поскольку смещения могут перекрываться, данная пара сегмент: смещение не является уникальной; следующие адреса относятся к одной и той же точке памяти:

0000: 0123

0002: 0103

0008: 00A3

0010: 0023

0012: 0003

Сегменты могут (но не должны) перекрываться. Например, все четыре сегмента могут начинаться с одного и того же адреса, что означает, что вся ваша программа в целом займет не более 64 Кб - но тогда в пределах этой памяти должны поместиться и коды программы, и данные, и стек.

## 2.1 Виды моделей памяти

В 16-разрядных программах вы можете использовать 6 моделей памяти: крохотную, малую, среднюю, компактную, большую и огромную.

Tiny (крохотная). Эта модель памяти используется в тех случаях, когда абсолютным критерием достоинства программы является размер ее загрузочного кода. Это минимальная из моделей памяти. Все четыре сегментных регистра (CS, DS, SS и ES) устанавливаются на один и тот же адрес, что дает общий размер кода, данных и стека, равный 64К. Используются исключительно ближние указатели.

Small (малая). Эта модель хорошо подходит для небольших прикладных программ. Сегменты кода и данных расположены отдельно друг от друга и не перекрываются, что позволяет иметь 64К кода программы и 64К данных и стека. Используются только указатели near.

Medium (средняя). Эта модель годится для больших программ, для которых не требуется держать в памяти большой объем данных. Для кода, но не для данных используются указатели far. В результате данные плюс стек ограничены размером 64К, а код может занимать до 1М.

Compact (компактная). Лучше всего использовать эту модель в тех случаях, когда размер кода невелик, но требуется адресация большого объема данных. Указатели far используются для данных, но не для кода. Следовательно, код здесь ограничен 64К, а предельный размер данных - 1 Мб.

Large (большая). Модели large и huge применяются только в очень больших программах. Дальние указатели используются как для кода, так и для данных, что дает предельный размер 1 Мб для обоих.

Huge (огромная). Дальние указатели используются как для кода, так и для данных. Borland C обычно ограничивает размер статических данных 64К; модель памяти huge отменяет это ограничение, позволяя статическим данным занимать более 64К.

Для выбора любой из этих моделей памяти вы должны либо воспользоваться соответствующим параметром меню интегрированной среды, либо ввести параметр при запуске компилятора, работающего в режиме командной строки.

## 2.2 Размещение исполняемого файла в ОЗУ в модели large

Модели памяти устроены по-разному. Рассмотрим расположение областей памяти в модели large.

Область кода содержит машинные коды функций программы. Функции, присоединенные к exe-файлу на стадии линковки, размещаются вне области кода.

Область данных содержит глобальные и статические переменные, строковые константы.

В стеке размещаются локальные переменные, параметры, передаваемые функциям, и ряд других данных. Как правило, стек растет сверху вниз, занимая пульсирующую непрерывную область. В случае переполнения стека происходит его "налезание" стека на область данных и выдается соответствующее сообщение. Проверка стека увеличивает время работы программы и ее можно отключить в Options-Entry/Exit Code Generation-Stack options-Test stack overflow.

В кучу данные помещаются только по указанию программиста и не имеют имени. К ним можно обратиться только по адресу, расположенному в локальной или глобальной переменной.

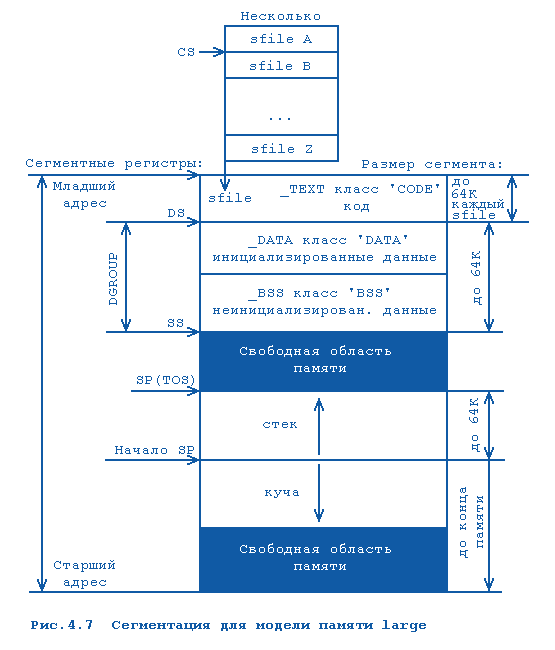


Рис.1. Сегментация для модели памяти Large

## 3. Передача данных в функцию

Си - язык сугубо процедурный и основной логической единицей программы является функция. Формат описания функции следующий:

[тип возвращаемого значения] имя\_функции (список параметров)

{тело функции

[return возвращаемое\_значение] }

В скобках помещена необязательная часть конструкции.

В списке параметров указывают данные, которые необходимо передать в функцию. Ниже рассмотрены различные способы передачи данных в функцию.

## 

## 3.1 Передача параметров по значению

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются. Поскольку передача параметров происходит по значению, в теле функции нельзя изменить значения переменных в вызывающей функции, являющихся фактическими параметрами.

Например:

void print\_num (int i, int j)

{ printf ("значение i=%d. Значение j=%d. ", i,j);}

Обращение в программе к данной функции будет таковым:

print\_num (6, 19);

## 3.2 Передача параметров по адресу

Рассмотрим пример функции, которая меняет значение переменных местами:

void change (int x, int y)

{ int k=x;

x=y;

y=k;

}

В данной функции значения переменных x и y, являющихся формальными параметрами, меняются местами, но поскольку эти переменные существуют только внутри функции change, значения фактических параметров, используемых при вызове функции, останутся неизменными. Для того чтобы менялись местами значения фактических аргументов можно использовать функцию приведенную в следующем примере.

Пример:

void change (int \*x, int \*y)

{ int k=\*x;

\*x=\*y;

\*y=k;

}

При вызове такой функции в качестве фактических параметров должны быть использованы не значения переменных, а их адреса change (&a,&b);

## 3.3 Передача одномерных массивов

При передаче одномерного массива в функцию следует учитывать, что имя массива не содержит информации о размере этого массива. Поэтому необходимо передавать два параметра: имя массива и размер.

Пример.

int sum (int A [], int Dim); // прототип

int sum (int A [], int Dim); // заголовок

{

….

} // телофункции

void main ()

{

int res, A [] = {2,1,3,2};

res = sum (A,

4);

} // вызов функции sum

Формальный аргумент имени массива может иметь вид int \*A.

int sum (int \*A, int Dim); // *прототип*

Для определения размера массива при вызове функции можно использовать выражение *sizeof (A) /sizeof (int) или sizeof (A) /sizeof (A []).* Например,

res = sum (A, sizeof (A) /sizeof (A []));

## 3.4 Передача двумерных массивов

Способ 1. При передаче двумерного массива в функцию следует учитывать, что количество элементов в строке массива является частью типа имени этого массива. Так например, для массива *int A [3] [4]* имя массива А имеет тип *int (\*) [4],* т.е. *А* - это указатель на одномерный массив из *4* элементов типа *int*.

Поэтому необходимо передавать два параметра: имя массива и количество строк в массиве.

Пример.

int sum2 (int A [] [4], int M); // прототип

int sum2 (int A [] [4], int M); // заголовок

{

….

} // телофункции

void main ()

{

int res, A [] [4] = {{2,1,3,2}, {2,3,4,5}}; // две строки, четыре столобца

res = sum2 (A,

2); // вызов функции sum2

}

Формальный аргумент имени массива может иметь вид int (\*A) [4].

int sum2 (int (\*A) [4], int M); // прототип

Для определения размера массива при вызове функции можно использовать выражение sizeof (A) /sizeof (A []). Например,

res = sum2 (A, sizeof (A) /sizeof (A []));

Способ 2. Другой способ передачи двумерного массива в функцию состоит в погружении фактического двумерного массива с размерами MxN в двумерный массив заведомо больших размеров. Пи этом достаточно взять большой размер стоки, например, 100.

int sum3 (int A [] [100], int M, int N); // прототип

int sum3 (int A [] [100], int M, int N); // заголовок

{

….

} // телофункции

void main ()

{

int res, A [2] [100] = {{2,1,3,2}, {2,3,4,5}}; /\* две строки, четыре столбца с чатичной инициалиазацией\*/

res = sum3 (A, 2,4); // вызов функции sum3

}

Способ 3. Третий способ передачи двумерного массива в функцию состоит в эмуляции фактического двумерного массива с размерами MxN с помощью одномерного массива с размером M\*N. Пи этом M\*N должно быть меньше 64К.

int sum4 (int A [], int M, int N); // прототип

int sum4 (int A [], int M, int N); // заголовок

{

….

} // телофункции

void main ()

{

int res, A [2] [4] = {{2,1,3,2}, {2,3,4,5}}; /\* две строки, четыре столбца \*/

res = sum4 ( (int \*) A, 2,4); // вызов функции sum4

}

## 4. Тестирование функций

Вычислительные модули необходимо тщательно протестировать с помощью отдельной тестовой функции с прототипом

void test (void);

При тестировании следует соблюдать следующие требования:

автоматизм, то есть от программиста при тестировании не требуется никаких действий,

прозрачность. Это означает, что функция test выводит сообщения на экран только в случае возникновения ошибок.

иллюстративность: листинг тестовой функции позволяет посмотреть различные способы вызова проверяемой функции.

всесторонность, то есть при тестировании необходимо рассмотреть все крайние ситуации.

В программе должны осуществляться все возможные проверки, в частности:

на корректность входных данных,

при выделении динамической памяти,

на выход индексов массива из диапазона.

## 5. Практические задания

## 5.1 Указать классы памяти переменной

Что напечатает программа? Укажите область действия, область видимости и продолжительность жизни всех переменных n.

int n = 1;

void main ()

{

printf (“%d", n);

static int n=3;

printf (“%d", n);

while (n--)

{

printf (“%d", n);

int n=10;

printf (“%d", n);

printf (“%d”,:: n+n);

}

}

## 5.2 Работаем с адресами

У некоторой программы в модели large при работе в отладчике регистры содержат следующие значения:

CS = 1ADF, DS=1AE3, SS=1B26, SP=0FD2.

Найдите размеры областей памяти.

Укажите диапазоны возможных адресов для:

переменной int n=2, если она а) глобальная, б) статическая, в) локальная;

содержимого указателя char \*str=”Hello”;

значения адресной константы main;

содержимого указателя int \*A= (int \*) malloc (1000).

## 5.3 Прототипы функций

## 5.3.1 Свопинг

Напишите прототип функции, которая организует обмен значений двух переменных.

## 5.3.2 Индексы максимальных элементов одномерного массива

Напишите прототип функции, которая находит индексы максимальных элементов одномерного массива.

## 5.3.3 Индексы максимальных элементов двумерного массива

Напишите прототип функции, которая находит индексы максимальных элементов двумерного массива с заданными размерами.

## 5.3.4 Угол между двумя векторами

Напишите прототип функции, которая находит угол в радианах между двумя векторами из пространства Rn.

## 5.3.5 Определитель матрицы

Напишите прототип функции, которая находит определитель квадратной матрицы с размерами nxn.

## 5.4 Выделение фрагмента программы в отдельную функцию

Разбиение программы на функции

#include <stdio. h>

void main ()

{

int a, b, div, mod;

printf (“Введите два целых числа”);

scanf (“%d%d”, &a, &b);

div = a/b;

mod = a%b;

printf (“\n%d /%d =%d”, a, b, div);

printf (“\n%d%%%d =%d", a, b, mod);

}

Выделите в три отдельных функции фрагменты, связанные с вводом данных, с вычислениями и с организацией вывода результатов. Вычислительная функция должна быть одна. Глобальные переменные не использовать.

## 5.5 Тестирование функции

## 5.5.1 Сортировка массива

Следующий прототип функции сортирует массив А размером n по возрастанию на месте оригинального массива void sort (int A [], int n);

Напишите тест этой функции для трех различных вариантов исходных данных.

## 5.5.2 МиниМакс

Функция находит минимальное и максимальное из двух чисел типа int и имеет прототип

void MinMax (int a, int b, int \*pmin, int \*pmax);

Напишите тест для этой функции.

## 6. Лабораторные задания

## 6.1 Линейное уравнение

Написать функцию, которая решает линейное уравнение a ∙ x + b = 0 с проверкой выхода за диапазон типа float. Прототип функции

int linur (float a, float b, float \*px);

Функция получает: a и b - коэффициенты уравнения, px - указатель на ячейку, в которую будет помещен единственный корень уравнения.

Функция возвращает:

0 - нет решения,

1 - найдено единственное решение,

2 - любое число является решением,

3 - решение единственное, но не входит в диапазон типа переменной x.

Вывод текстовой информации с результатами решения организовать с использованием оператора switch. Выход из программы должен быть единственным. Организовать тестирование функции linur.

## 6.2 Четные элементы массива

Напишите функцию, которая находит все четные элементы одномерного массива элементов типа int. Организовать тестирование функции.

Прототип функции

int Chot (int A [], int DimA, int FoundA []);

## 6.3 Нахождение простых чисел

Напишите функцию, которая находит все простые числа и их количество до long N включительно. Натуральное число m > 1 называется простым, если оно делится только на 1 и на само себя. Организовать тестирование функции.

Прототип функции.

long AllProst (long N, long Prost [], int DimProst, int \*flag);

## 6.4 Количество вхождений подстроки в строку

Напишите функцию, которая определяет количество вхождений подстроки в строку. Организовать тестирование функции.

Прототип функции

int NumStrStr (char \*str, char \*substr);

## 6.5 Произведение матриц

Напишите функцию, которая находит произведение двух прямоугольных матриц с согласованными размерами. Организовать тестирование функции.

Прототип функции

void MMult (float A [], float B [], float AB [], int m, int n, int k);

Здесь одномерные массивы эмулируют двумерные массивы.

## 7. Дополнительные задания

Написать функцию принадлежности точки невыпуклому многоугольнику без самопересечений.

Написать функции tolowerrus и toupperrus для перевода одной русской буквы из верхнего регистра в нижний и наоборот.

## Библиографический список

1. Керниган Б. Язык программирования Си / Б. Керниган, Д. Ритчи. СПб.: Невский диалект, 2001.352 с.
2. Подбельский В.В. Программирование на языке Си / В.В. Подбельский, С.С. Фомин. М.: Финансы и статистика, 2004.600 с.
3. Программирование в Си. Организация ввода-вывода: метод. указания / сост. С.П. Трофимов. Екатеринбург: УГТУ, 1998.14 с.
4. Программирование в Си. Динамическое распределение памяти: метод. указания / сост. С.П. Трофимов. Екатеринбург: УГТУ, 1998.13 с.