Лабораторная работа 8

Обработка и печать числовой матрицы.

Содержание работы:
- создание двухмерных динамических массивов;
- обработка матриц;
-использование файлов для хранения матриц (на примере приведенной ниже программы сортировки строк матрицы);
- передача двумерных массивов в функцию через параметры;
- форматированный вывод матриц на экран;
- доступ к элементам матрицы через указатели и с помощью индексов;
- освоение технологии структурного программирования.

Задание. Создать квадратную матрицу A размером N*N (где N вводится с клавиатуры), и заполнить её следующими значениями:
- все элементы главной диагонали равны 1;
- элементы, лежащие выше главной диагонали, вычисляются по формуле
$\mathbf{A_{i,j} = x^i / (j!)^i}$, а элементы, лежащие ниже главной диагонали, по формуле
$\mathbf{A_{i,j} = (-x)^i / (j!)^i}$, где $\mathbf{i,j = 1,2,\ldots,N}$.
Для вычисления значений элементов матрицы использовать рекуррентные соотношения**.**

Реализовать алгоритм заполнения матрицы в виде функции.

В зависимости от размера матрицы и ширины поля вывода элемента матрицы, обеспечить удобное для пользователя отображение матрицы на экране. Оформить вывод матрицы размером N*M на экран в виде функции с целью использования ее в последующих лабораторных работах для распечатки двумерных массивов.

Матрица должна передаваться в разрабатываемые функции через параметры.

Не изменяя кода функции вывода матрицы, распечатать матрицу в «научном» формате и в формате с фиксированной точкой с точностью 8 знаков после запятой.

Распечатать с помощью разработанной функции, используя вспомогательный массив указателей на строки, матрицу размером **B[10][10],** заданную с помощью оператора описания (нединамическую). Значение элементов матрицы **B** определяется соотношением: **B[i][j]=i*10+j**.

Объясните, как передаются матрицы A и B в функцию вывода матриц на экран.

Вставьте в программу и объясните результаты выполнения следующих операторов для матрицы B[10][10]:

```
cout<<B<<"  "<<B[0]<<"  "<<B[2]<<endl;
cout<<B[0][0]<<"  "<<**B <<"  "<<*B[0]<<endl;
cout<<*(*(B+1))<<"  "<<*B[1]<<endl;
cout<<*(B[0]+1)<<"  " <<*(*B+1)<<endl;
cout<<B[0][20]<<"  "<<*(B[0]+20)<<"  "<<*B[2]<<endl;
```

Прежде чем приступать к выполнению задания прочитайте приведенный ниже текст и разберите и выполните пример программы сортировки строк матрицы.

Создание двухмерных динамических массивов.

В динамической области памяти можно создавать двумерные массивы с помощью операции new или функции malloc. Остановимся на первом варианте, поскольку он более безопасен и прост в использовании.

Обращение к элементам динамических массивов производится точно так же, как к элементам «обычных», с помощью конструкции вида a[i][j].

Универсальный способ выделения памяти под двумерный массив, когда обе его размерности задаются на этапе выполнения программы, приведен ниже:

```
int nrow, ncol;
cout << " Введите количество строк и столбцов :";
cin >> nrow >> ncol;
int **a = new int *[nrow];                    // 1
```

```
for(int i = 0; i < nrow; i++)                    // 2
a[i] = new int [ncol];                           // 3
```

Для того чтобы понять, отчего динамические массивы описываются именно так, нужно разобраться в механизме индексации элемента массива. Поскольку для доступа к элементу массива применяется две операции разадресации, то переменная, в которой хранится адрес начала массива, должна быть указателем на указатель.

В операторе 1 объявляется переменная типа «указатель на указатель на **int**» и выделяется память под массив указателей на строки массива (количество строк — **nrow**). В операторе 2 организуется цикл для выделения памяти под каждую строку массива. В операторе 3 каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива. Каждая строка состоит из **ncol** элементов типа **int** (рис. 1).

Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции **delete [],** например: delete [] a;

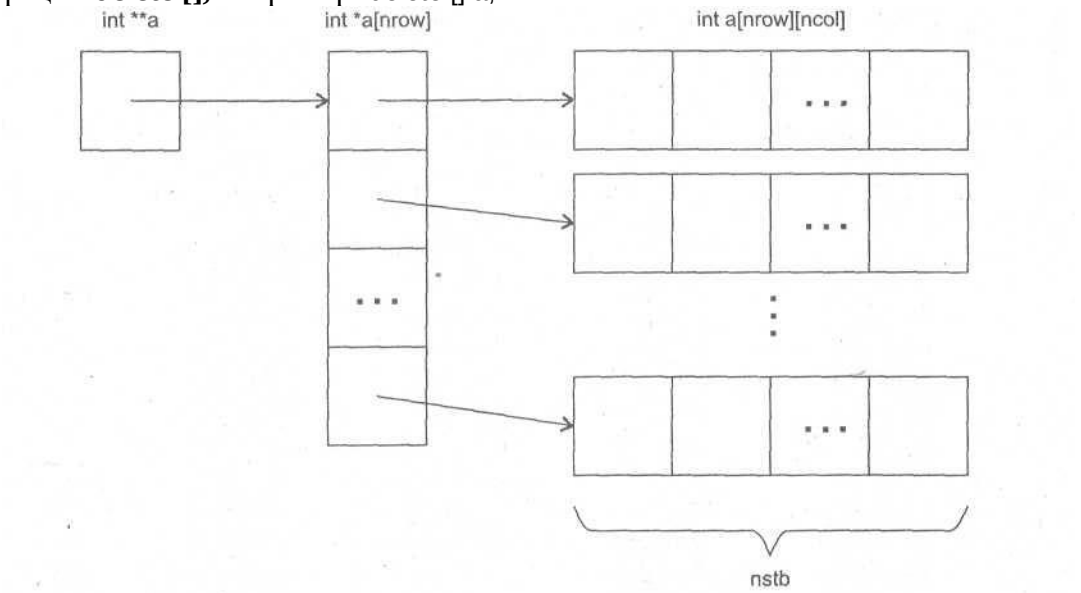

Рис. 1. Схема динамической области памяти, выделяемой под массивы

Передача многомерного массива в функцию с помощью параметров.

При необходимости передать в функцию многомерный массив с помощью параметра возникают неудобства, связанные с отсутствием в Си++ и Си объектов типа многомерный массив. Если мы описываем массив с несколькими индексами, например,

double arr[6][4][2];

то это не трехмерный массив, а одномерный массив с именем arr, состоящий из **6** элементов, каждый из которых имеет тип **double [4][2].** В свою очередь, каждый из этих элементов есть одномерный массив из четырех элементов типа **double [2].** И, наконец, каждый из этих элементов является массивом из двух элементов типа **double**.

Очевидное и неверное решение при попытке передать в функцию матрицу – определить её заголовок следующим образом:

void func(double x[][], int n)

Здесь **n** – предполагаемый порядок квадратной матрицы; **double x[][]** – попытка определить двухмерный массив с заранее неизвестными параметрами. На такую попытку транслятор ответит сообщением об ошибке:

Error…: Size of type is unknown or zero.

Вспомним – массив всегда одномерный, а его элементы должны иметь известную и фиксированную длину. В массиве **double x[][]** не только неизвестно количество элементов одномерного массива (это допустимо и их можно передать параметром **int n),**

но ничего не известно о размерах этих элементов. Допустимое с точки зрения синтаксиса языка Си++ решение - **void func(double x[][4], int n).**

Нежизненность такого решения – необходимость фиксации второй размерности матрицы.

Указанные ограничения на возможность применения многомерных массивов в качестве параметров функции можно обойти двумя путями.

Первый путь – подмена многомерного массива, например, **double x[3][4]** одномерным **double x[12]** и имитация внутри функции доступа к нему как к многомерному массиву.

Второй путь – использование вспомогательных одномерных массивов указателей на массивы. Такой массив указателей на строки матрицы используется при создании динамических массивов.

Помните, что если размерность массива явно не указана, то в функцию с помощью параметров можно передавать только одномерные массивы.

Программа сортировки строк матрицы.

Написать программу, которая упорядочивает строки прямоугольной целочисленной матрицы по возрастанию сумм их элементов.

Давайте на этом примере формализуем общий порядок создания структурной программы, которому мы ранее следовали интуитивно. Этого порядка полезно придерживаться при решении даже простейших задач

I. ***Исходные данные, результаты и промежуточные величины.*** Как уже неоднократно упоминалось, начинать решение задачи необходимо с четкого описания того, что является ее исходными данными и результатами и каким образом они будут представлены в программе.

Исходные данные. Поскольку размерность матрицы неизвестна, придется использовать динамический массив элементов целого типа. Ограничимся типом int, хотя для общности следовало бы воспользоваться максимально длинным целым типом.

Результаты. Результатом является та же матрица, но упорядоченная. Это значит, что нам не следует заводить для результата новую область памяти, а необходимо упорядочить матрицу на том же месте. В данной задаче такое требование может показаться излишним, но в общем случае, когда программист работает в команде и должен передавать результаты коллеге, это важно. Представьте себе ситуацию, когда коллега думает, что получил от вас упорядоченную матрицу, а на самом деле вы сформировали ее в совершенно другой области памяти.

Промежуточные величины. Кроме конечных результатов, в любой программе есть промежуточные, а также служебные переменные. Следует выбрать их тип и способ хранения.

Очевидно, что если требуется упорядочить матрицу по возрастанию сумм элементов ее строк, эти суммы надо вычислить и где-то хранить. Поскольку все они потребуются при упорядочивании, их надо записать в массив, количество элементов которого соответствует количеству строк матрицы, а i-й элемент содержит сумму элементов i-й строки. Количество строк заранее неизвестно, поэтому этот массив также должен быть динамическим. Сумма элементов строки может превысить диапазон значений, допустимых для отдельного элемента строки, поэтому для элемента этого массива надо выбрать тип 1ong.

После того как выбраны структуры для хранения данных, можно подумать и об алгоритме (именно в таком порядке, а не наоборот — ведь алгоритм зависит от того, каким образом представлены данные).

II. Алгоритм работы программы. Для сортировки строк воспользуемся одним из самых простых методов — методом выбора. Он состоит в том, что из массива выбирается наименьший элемент и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом и так далее n - 1 раз. Одновременно с обменом элементов массива выполняется и обмен значений двух соответствующих строк матрицы.

Алгоритм сначала записывается в самом общем виде (например, так, как это сделано выше). Пренебрегать словесным описанием не следует, потому что процесс формулирования на естественном языке полезен для более четкого понимания задачи. При этом надо стремиться разбить алгоритм на простую последовательность шагов. Например, любой алгоритм можно первоначально разбить на этапы ввода исходных данных, вычислений и вывода результата.

Вычисление в данном случае состоит из двух шагов: формирование сумм элементов каждой строки и упорядочивание матрицы. Упорядочивание состоит в выборе наименьшего элемента и обмене с первым из рассматриваемых. Разветвленные алгоритмы и алгоритмы с циклами полезно представить в виде обобщенной блок-схемы.

III. Кодирование. Когда алгоритм полностью прояснился, можно переходить к написанию программы. Одновременно с этим продумываются и подготавливаются тестовые примеры. Не ленитесь придумать переменным понятные имена и сразу же при написании аккуратно форматировать текст программы, чтобы по положению оператора было видно, на каком уровне вложенности он находится. Функционально завершенные части алгоритма отделяются пустой строкой, комментарием или хотя бы комментарием вида

//---

IV. Отладка. При написании программы рекомендуется всегда включать в нее промежуточную печать вычисляемых величин в удобном для восприятия формате. Это простой и надежный способ контроля хода выполнения программы.

Не нужно стремиться написать сразу всю программу. Сначала пишется и отлаживается фрагмент, содержащий ввод исходных данных. Затем промежуточную печать можно убрать и переходить к следующему функционально законченному фрагменту алгоритма. Для отладки полезно выполнять программу по шагам с наблюдением значений изменяемых величин и сравнением их с контрольным примером.

Программа сортировки строк матрицы.

```
#include <fstream.h>
#include <iomanip.h>
//using namespace std;
int main()
{
ifstream fin("input.txt", ios::in);
if (!fin)
{ cout <<" Файл input.txt не найден " << endl; return 1; }
int nrow,ncol;
fin>>nrow>>ncol;     //    ввод размерности массива
int i, j;
// выделение памяти под вспомогательный массив указателей на строки
int **a = new int *[nrow];
// выделение памяти под строки матрицы
//и инициализация вспомогательного массива указателей на строки
for(i = 0; i < nrow; i++)
      a[i] = new int [ncol];
//ввод массива
for (i = 0; i < nrow; i++)
    for (j = 0; j < ncol; j++)
        fin >> a[i][j];
// формирование массива сумм элементов строк
long *sum = new long [nrow];
for (i =0; i < nrow; i++)
{
    sum[i] = 0;
    for (j = 0; j < ncol; j++)
        sum[i] += a[i][j];
}
// контрольный вывод
for (i = 0; i < nrow; i++)
{
    for (j = 0; j < ncol; j++)
        cout << setw(4) << a[i][j] << " ";
    cout << "| " << sum[i] << endl;
}
cout << endl;
long buf_sum;
int nmin, buf_a;
for (i = 0; i < nrow - 1; i++) // упорядочивание
{
    nmin = i;
    for (j = i + 1; j < nrow; j++)
        if (sum[j] < sum[nmin]) nmin = j;
           buf_sum = sum[i];
    sum[i] = sum[nmin];
    sum[nmin] = buf_sum;
    for (j = 0; j < ncol; j++)
    {
        buf_a = a[i][j];
```

```
        a[i][j] = a[nmin][j];
        a[nmin][j] = buf_a;
    }
}
for (i = 0; i < nrow; i++)
{ // вывод упорядоченной матрицы
     for (j = 0; j < ncol; j++) cout <<setw(4) << a[i][j] << " ";
          cout << endl;
}
return 0;
}
```

В программе используются две буферные переменные: buf_sum, через которую осуществляется обмен двух значений сумм, имеет такой же тип, что и сумма, а для обмена значений элементов массива определена переменная buf_a того же типа, что и элементы массива.

Как и в предыдущем примере, данные читаются из файла. Этот способ ввода является предпочтительнее стандартного ввода, поскольку при формировании файла легче продумать, какие значения лучше взять для исчерпывающего тестирования программы. В данном случае для первого теста следует подготовить массив не менее чем из четырех строк с небольшими значениями элементов для того, чтобы можно было в уме проверить, правильно ли вычисляются суммы.

Ввод размерности массива и его элементов выполняется из файла input.txt, расположенного в том же каталоге, что и программа, а результаты выводятся в файл output. txt. В программе определены объект fin класса входных файловых потоков и объект fout класса выходных файловых потоков. Файловые потоки описаны в заголовочном файле <fstream.h>. Работа с этими объектами аналогична работе со стандартными объектами cin и cout, то есть можно пользоваться теми же операциями помещения в поток << и извлечения из потока >>.

Предполагается, что файл с именем input.txt находится в том же каталоге, что и текст программы, иначе следует указать полный путь, дублируя символ обратной косой черты, так как иначе он будет иметь специальное значение:

```
ifstream fin("c:\\A_Worker\\input.txt", ios::in | ios::nocreate);
```

Обратите внимание, что для контроля вместе с исходным массивом рядом с каждой строкой выводится сумма ее элементов, отделенная вертикальной чертой.

Дополнительно следует проверить, правильно ли упорядочивается массив из одной и двух строк и столбцов, поскольку многие ошибки при написании циклов связаны с неверным указанием их граничных значений.

Основные правила работы с двухмерными массивами.

1. В массивах, определенных с помощью операторов описания, обе размерности должны быть константами или константными выражениями.
2. Массив хранится по строкам, каждая строка хранится в непрерывной области памяти.
3. Первый индекс всегда представляет собой номер строки, второй — номер столбца. Каждый индекс может изменяться от 0 до значения соответствующей размерности, уменьшенной на единицу.
4. При описании массива можно в фигурных скобках задать начальные значения его элементов.
5. Для выделения динамической памяти под массив, в котором все размерности переменные, используются циклы.
6. Освобождение динамической памяти из-под массива с любым количеством измерений выполняется с помощью операции **delete [ ] A**, где **A** – имя массива.

Рекомендации по порядку создания программы.

1. Выбрать тип и способ хранения в программе исходных данных, результатов и промежуточных величин.

2. Записать алгоритм сначала в общем виде, стремясь разбить его на простую последовательность шагов, а затем детализировать каждый шаг.

3. Написать программу. При написании программы рекомендуется:

- давать переменным понятные имена;
- не пренебрегать содержательными комментариями;
- использовать промежуточную печать вычисляемых величин в удобном формате;
- при написании вложенных циклов следить за отступами;
- операторы инициализации накапливаемых в цикле величин задавать непосредственно перед циклом, в котором они вычисляются.

4. Параллельно с написанием программы задать тестовые примеры, которые проверяют все ветви алгоритма и возможные диапазоны значений исходных данных. Исходные данные удобнее формировать в файле (по крайней мере, при отладке), не забывая проверять в программе успешность его открытия.