Computer Science 1: Starting Computing
CSCI 1300

University of Colorado Boulder

Computer Science 1: Starting Computing CSCI 1300

Dr. Ioana Fleming
Spring 2019
Lecture 19

Reminders

Submissions:

- Homework 6: due Monday 3/4 at 6pm
- Homework 7: due Wednesday 3/13 at 6pm

Readings:

- Ch. 6 – Arrays – 2D
- Ch. 8 – Streams

To discuss:

- Global Variables

© traveler1116/iStockphoto.

Chapter Six: Arrays and Vectors

Topic 6

1. Arrays
2. Common array algorithms
3. Arrays / functions
4. Problem solving: adapting algorithms
5. Problem solving: discovering algorithms
6. 2D arrays
7. Vectors
8. Chapter Summary

Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout.

Such data sets commonly occur in financial and scientific applications.

An arrangement consisting of *tabular data (rows and columns* of values) is called:

a ***two-dimensional array***, or a ***matrix***

Two-Dimensional Array Example

Consider the medal-count data from the 2014 Winter Olympic skating competitions:

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Defining Two-Dimensional Arrays

C++ uses an array with *two* subscripts to store a *2D* array.

```
const int COUNTRIES = 8;
const int MEDALS = 3;
int counts[COUNTRIES][MEDALS];
```

An array with 8 rows and 3 columns is suitable for storing our medal count data.

Defining Two-Dimensional Arrays – Initializing

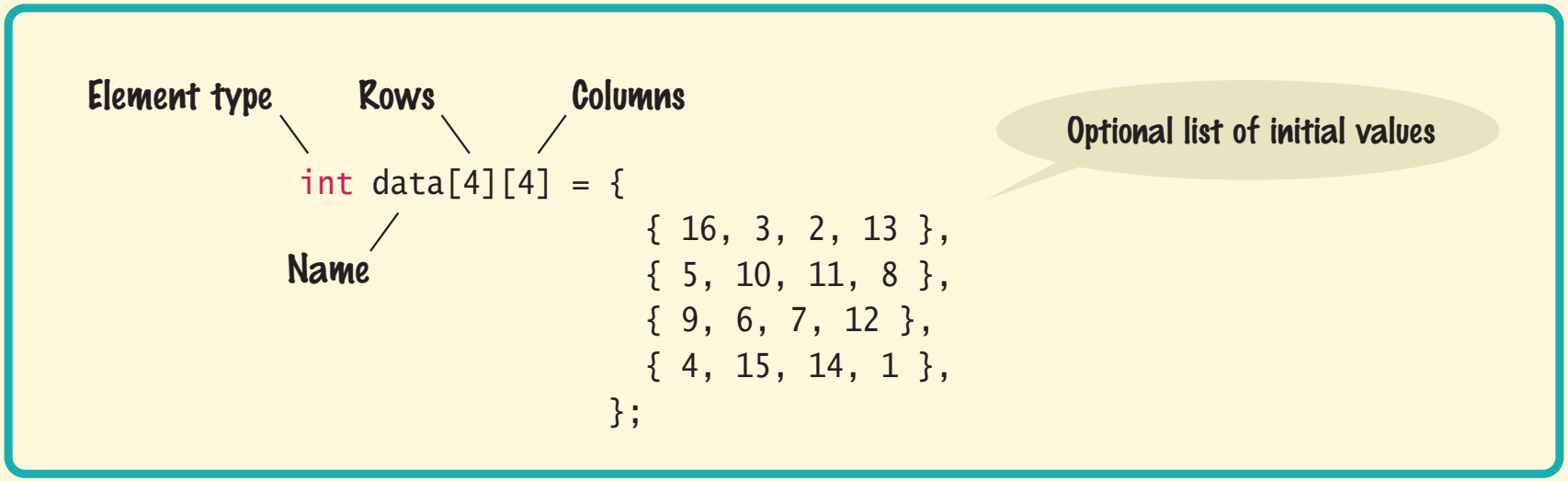
Just as with one-dimensional arrays, you *cannot* change the size of a two-dimensional array once it has been defined.

But you can initialize a 2-D array:

```
int counts[COUNTRIES][MEDALS] =
{
   { 0, 3, 0 },
   { 0, 0, 1 },
   { 0, 0, 1 },
   { 1, 0, 0 },
   { 0, 0, 1 },
   { 3, 1, 1 },
   { 0, 1, 0 }
   { 1, 0, 1 }
};
```

Defining 2D arrays

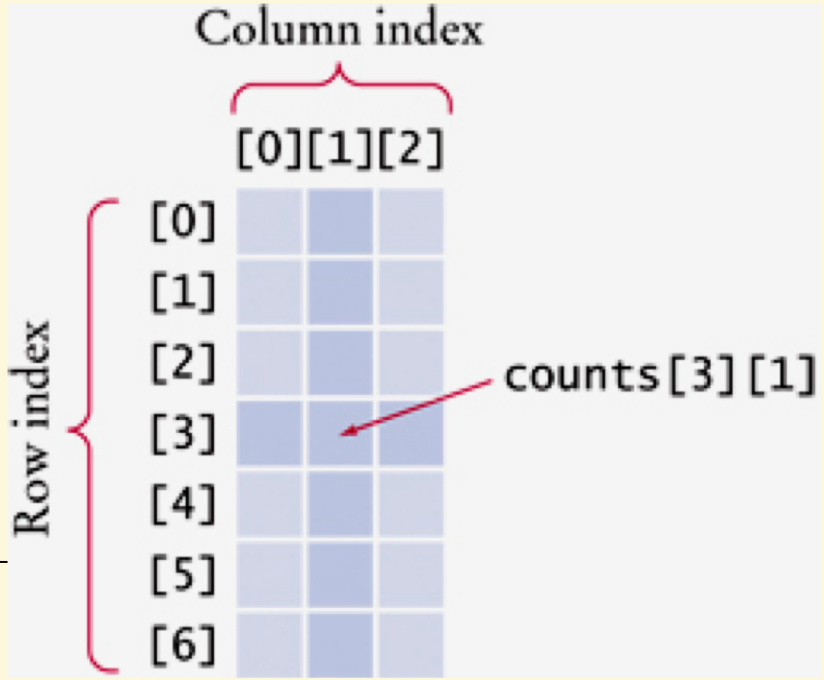
Two-Dimensional Array Definition



2D Arrays – Accessing Elements

```
// copy to value what is currently
// stored in the array at [3][1]
int value = counts[3][1];

// Then set that position in the array to 8
counts[3][1] = 8;
```



Two-Dimensional Array - Printing

Consider the medal-count data from the 2014 Winter Olympic skating competitions:

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

Print All Elements in a 2D Array

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

```
[0][0], [0][1], [0][2]
// Process the 1st row:
   for (int j = 0; j < MEDALS; j++)
   {
      cout << setw(8) << counts[0][j];
   }
```

Print All Elements in a 2D Array

Country	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

```
[0][0], [0][1], [0][2]
[1][0], [1][1], [1][2]
[2][0], [2][1], [2][2]
   for (int j = 0; j < MEDALS; j++)
   {
      cout << setw(8) << counts[i][j];
   }
```

Print All Elements in a 2D Array

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Process the ith row
   for (int j = 0; j < MEDALS; j++)
   {
      // Process the jth column in the ith row
      cout << setw(8) << counts[i][j];
   }
   // Start a new line at the end of the row
   cout << endl;
}
```

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter
  - 2nd dimension size IS given
- Example:

```
void DisplayPage(const char p[][100], int sizeDimension1)
{
    for (int index1=0; index1<sizeDimension1; index1++)
    {
           for (int index2=0; index2 < 100; index2++)
                  cout << p[index1][index2];
           cout << endl;
    }
}
```

2D Array Parameter Columns Hardwired

That function works for only arrays of 3 columns.

If you need to process an array
with a different number of columns, like 4,

you would have to write
a different function
that has 4 as the parameter.

Omitting the Column size of a 2D Array Parameter

When passing a one-dimensional array to a function, you specify the size of the array as a separate parameter variable:

```
void print(double values[], int size)
```

This function can print arrays of any size. However, for two-dimensional arrays you cannot simply pass the numbers of rows and columns as parameter variables:

```
void print(double table[][], int rows, int cols) //NO!
```

```
const int COLUMNS = 3;
void print(const double table[][COLUMNS], int rows) //OK
```

This function can print tables with any number of rows, but the column size is fixed.

Two-Dimensional Array Storage

What's the reason behind this?

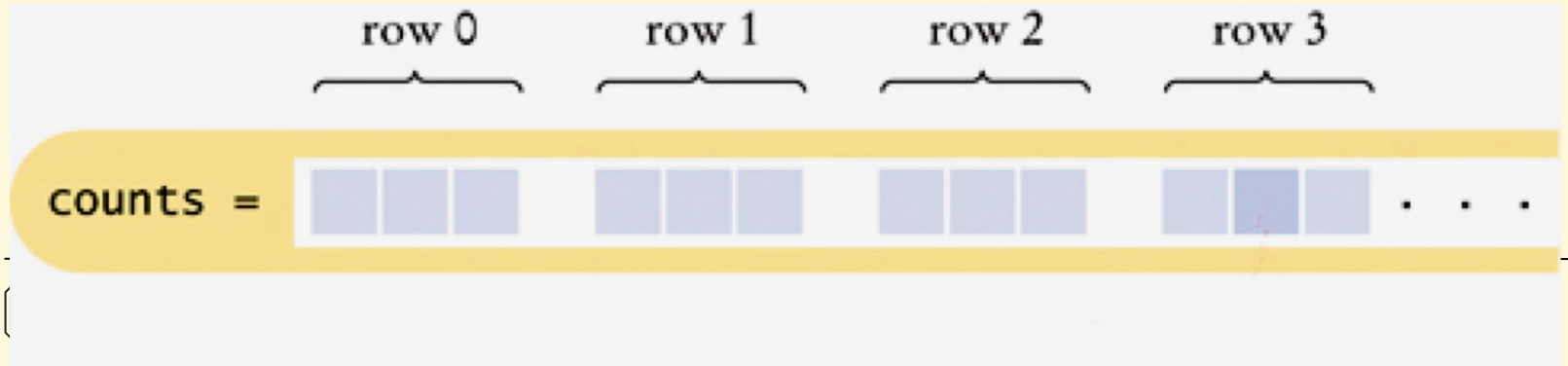
Although the array appears to be two-dimensional,
the elements are still stored as a linear sequence.

`counts` is stored as a sequence of rows, each 3 long.

So where is **`counts[3][1]`** ?

The offset (calculated by the compiler) from the start of the array is

`3 x number of columns + 1`



2D Array Parameters: Rows

The **`row_total`** function: does it need to know the number of rows of the array?

What about **`column_total()`?**

If the number of rows is required, pass it in:

```
int column_total(int table[][COLUMNS], int rows, int col)
{
   int total = 0;
   for (int i = 0; i < rows; i++)
   {
      total = total + table[i][col];
   }
   return total;
}
```

Practice It: 2D Array Parameters

Insert the missing statement. The function should return the result of adding the values in the first row of the 2D array received as argument.

```
int add_first_row(int array[][MAX_COLS], int rows, int
cols)
{
   int sum = 0;
   for (int k = 0; k < cols; k++)
   {
      sum = sum + ____________________;
   }
   return sum;
}
```

Arrays – Fixed Size is a Drawback

The size of an array *cannot* be changed after it is created.

You have to get the size right – *before* you define an array.

The compiler has to know the size to build it.
and a function must be told about the number
elements and possibly the capacity.

It cannot hold more than it's initial capacity.

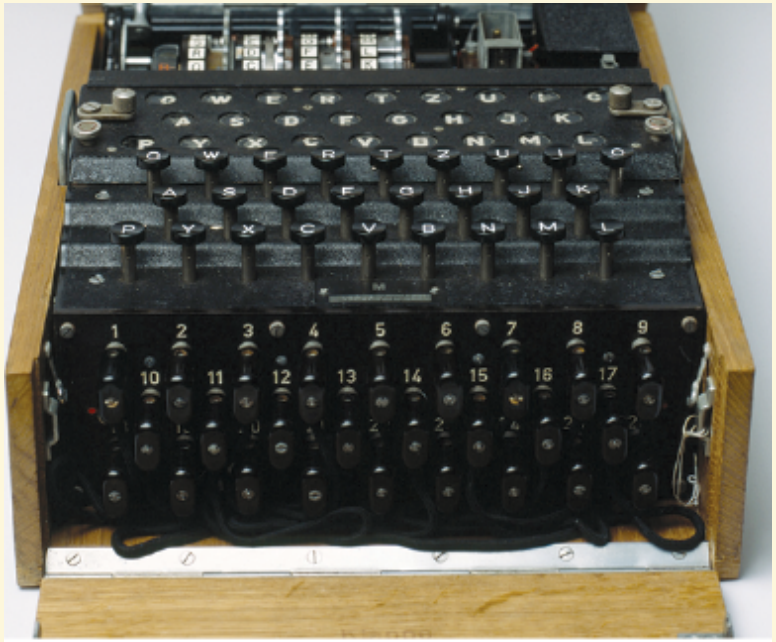
Later, we'll discuss vectors, which have variable size and some other programmer-friendly features lacking in arrays.

Summary 1

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind

Summary 2

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays → more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multidimensional arrays
  - Create "array of arrays"



James King-Holmes/Bletchley Park Trust/Photo Researchers, Inc.

Chapter Eight: Streams

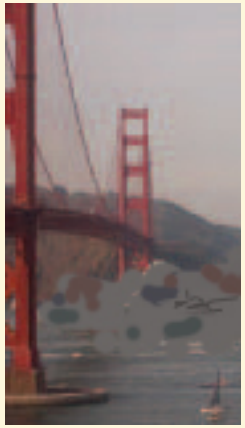
Chapter Goals

- To be able to read and write files
- To convert between strings and numbers using string streams

Topic 1

1. Reading and writing text files
2. Reading text input
3. Writing text output
4. Parsing and formatting strings
5. Command line arguments
6. Random access and binary files

Streams

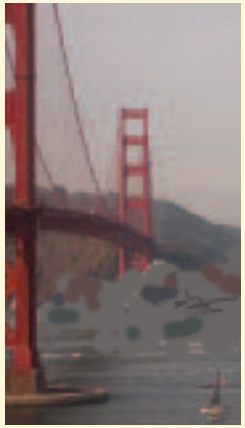


A very famous bridge

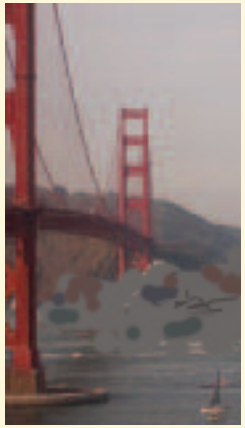
over a "*stream*"

Streams

A ship

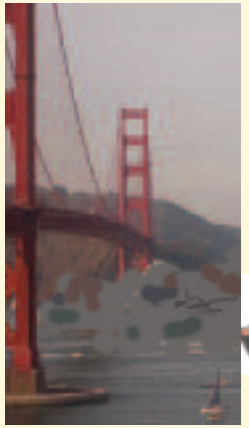


Streams



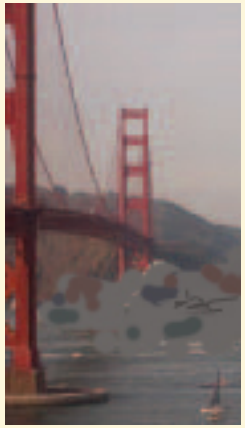
in the stream

Streams



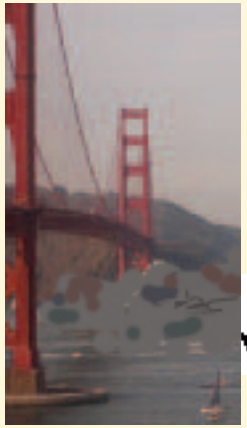
one at a time

Streams

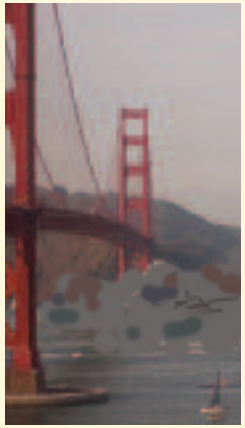

A ***stream*** of ships

Streams

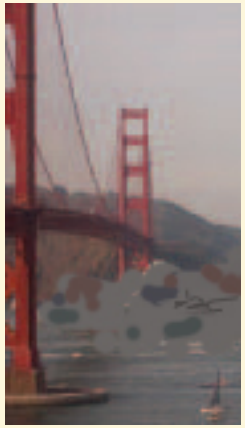


an *input stream* to that famous city

Streams



Streams



No more ships in the stream at this time

Let's process what we just input…

Reading and Writing Files

- The C++ input/output library is based on the concept of *streams*.
- An *input stream* is a source of data.
- An *output stream* is a destination for data.
- The most common sources and destinations for data are the files on your hard disk.
  - You need to know how to read/write disk files to work with large amounts of data that are common in business, administrative, graphics, audio, and science/math programs

Streams

```
This is a stream of characters. It could be from
the  keyboard  or  from a file. Each of these is
just a character - even these:  3  -23.73 which,
when input, can be converted to: ints or doubles or
whatever type you like.
(that was a  '\n'  at the end  of the last line)
&*@&^#!%#$ (No, that was -not- a curse!!!!!!!!!!
¥1,0000,0000 (price of a cup of coffee in Tokyo)
Notice that all of this text is very plain -  No
bold or green of italics - just characters - and
whitespace (TABs, NEWLINES and, of course... the
other one you can't see: the space character:
(another '\n')
(&& another)
```

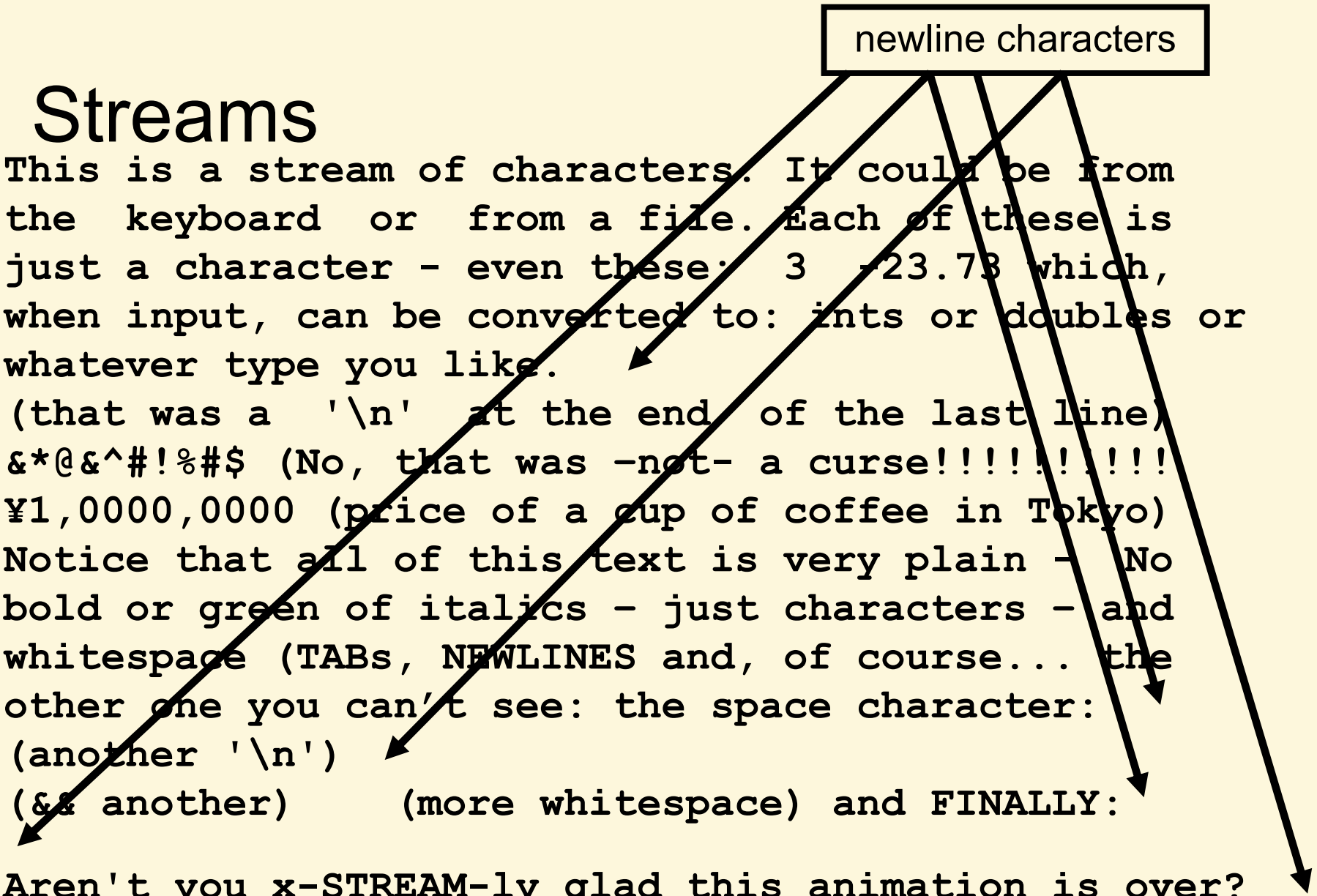
Reading and Writing Streams

The stream you just saw is a plain text file.
No formatting, no colors, no video or music

(or sound effects).

A program can read these sorts of plain text streams of characters from the keyboard, as has been done so far with `cin`.

Streams




```
This is a stream of characters. It could be from
the  keyboard  or  from a file. Each of these is
just a character - even these:  3  -23.73 which,
when input, can be converted to: ints or doubles or
whatever type you like.
(that was a  '\n'  at the end  of the last line)
&*@&^#!%#$ (No, that was -not- a curse!!!!!!!!!
¥1,0000,0000 (price of a cup of coffee in Tokyo)
Notice that all of this text is very plain -  No
bold or green of italics - just characters - and
whitespace (TABs, NEWLINES and, of course... the
other one you can't see: the space character:
(another '\n')
(&& another)    (more whitespace) and FINALLY:

Aren't you x-STREAM-ly glad this animation is over?
```

University of Colorado Boulder

And there were no sound effects!!!

Reading and Writing Disk Files

You can also read and write files stored on your hard disk:

- plain text files
- binary information (a binary file)
  - Such as images or audio recording

To read/write files, you use *variables* of the stream types:

`ifstream` for input from plain text files.

`ofstream` for output to plain text files.

`fstream` for input and output from binary files.

You must `#include <fstream>`