EEET2482
Software Engineering Design

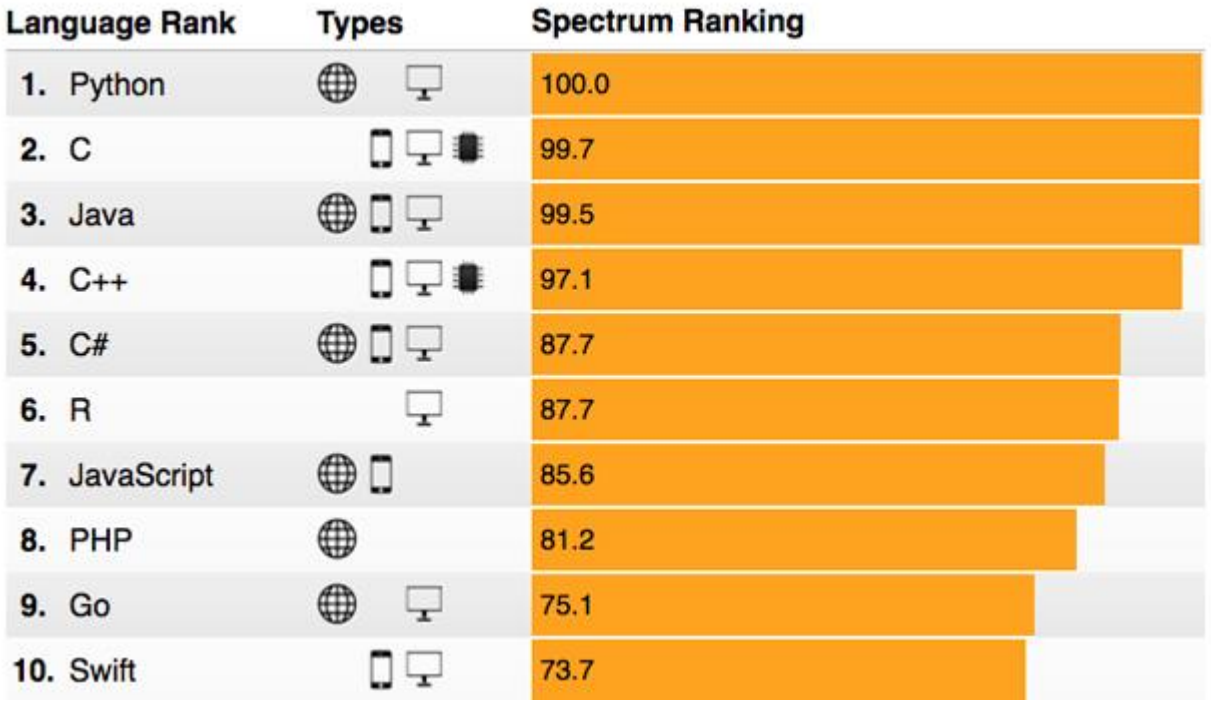
Module 2 – I/O Streams, File I/O and Strings

Lecturer: Mr. Linh Tran

Course note acknowledgments: Dr. Samuel Ippolito, E. Cheng, R. Ferguson, H. Rudolph, Pj. Radcliffe, G. Matthews.

The 2017 (*IEEE*)Top Ten Programming Languages

- What are the most popular programming languages?



- C/C++ still remain in the top 4, however have slipped form the number 1 and 2 place over the past few years due the popularity of python in the data sciences area
- C++ is increasing in popularity in the embedded area, however C still holds the number 1 spot for embedded

Source: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

Lecture Overview

- **C++ I/O System**
  - Input/Output Devices
- **C++ I/O Streams**
  - `cout`, `cin`, `cerr`, `clog` objects
  - Input / Output Library
  - Stream Formatting (Precision, width, fill, base, etc.)
    - `ios` class flags
    - I/O Manipulators
- **File I/O**
  - Opening a file
  - Writing to a text file
  - Reading from a text file
  - Binary file I/O
    - Single bytes: `get()`, `put()`
    - Working with Blocks of data
  - Using `getline()`
  - Difference between `FILE*` and `fstream` objects
- **String Class**
  - Quick introduction to string objects
  - A look at some useful functions

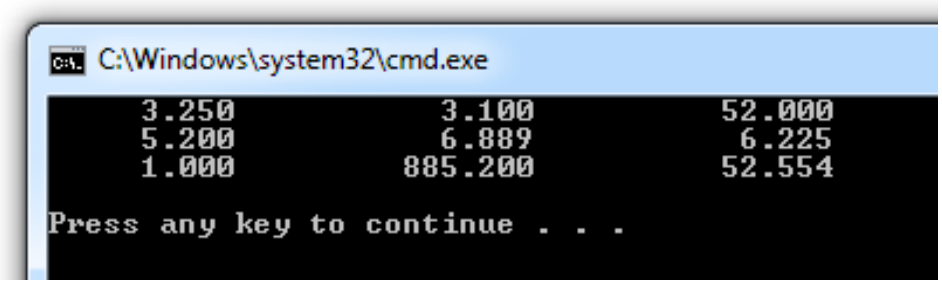
Consider the Following Problem

A 3x3 array of type double with variable precision:

```
double myarray[3][3] = { {3.25 , 3.1    , 52.0001      },
                         {5.2  , 6.8892 , 6.225        },
                         {1.0  , 885.2  , 52.554444444 }  };
```

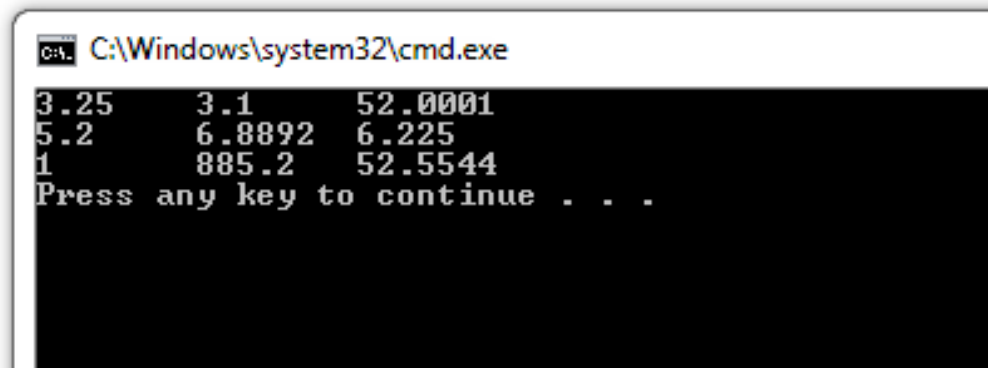
How to print these to the screen:

- Using evenly spaced columns
- 3 digits after the decimal point
- Decimal points should all line up



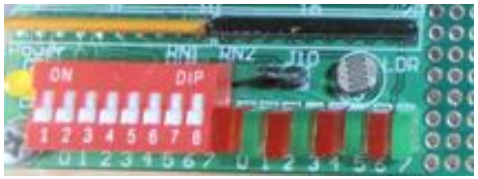
Using a simple for loop results in the following (messy output):

```
for(int i=0; i<3; i++)
{
   for(int j=0; j<3; j++)
   {
      cout << myarray[i][j] << "\t";
   }
   cout << "\n";
}
```



Input/Output Devices

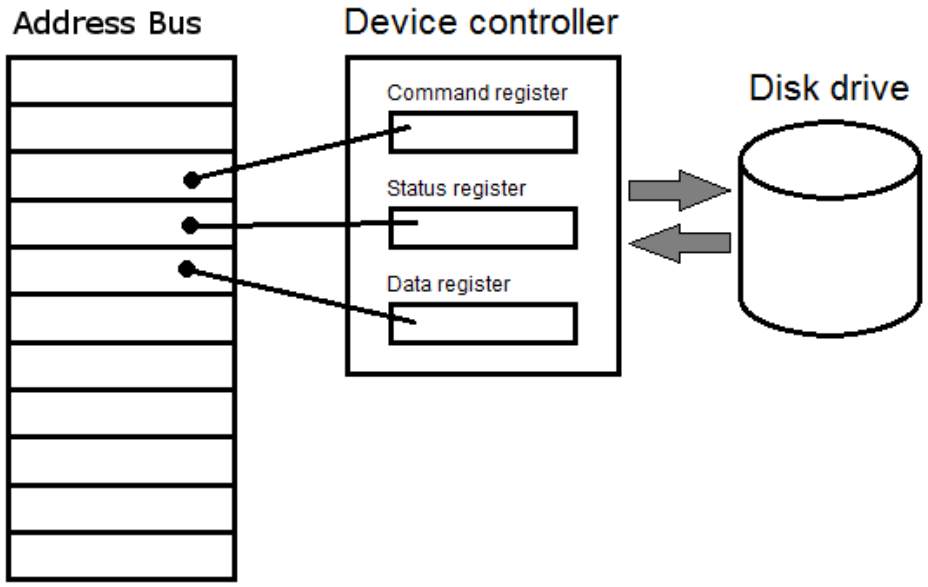
- All computers have input/output (I/O) devices (e.g. peripheral devices)
  - Some are **input** (read) only devices
    - Keyboards
    - Mouse
    - Scanners
  - Some are **output** (write) only devices
    - Printers,
    - Displays, LCD screens
    - Motor controls
    - Lights
  - Some are **input and output** devices
    - disk drive,
    - network connection,
    - USB port
  - Some are **read or write** (input or output) devices
    - e.g. a DVD writer
      - Read device if insert a prewritten DVD
      - Write device if insert a writeable DVD (and once written can read it)



PortC PortB

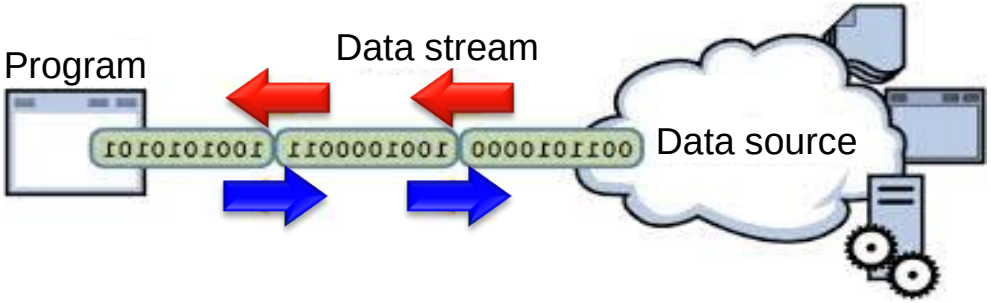
Memory Mapped I/O Register (MMIO)

- MMIO is the process of interacting with hardware by reading and writing from pre-defined memory addresses
- A peripheral device is an internal or external device which connects directly to the computer, but is not required for general computing operations
- Typically, a peripheral device is connected to one or more of the computer's memory addresses, where data transfer occurs by reading and writing from the associated memory addresses



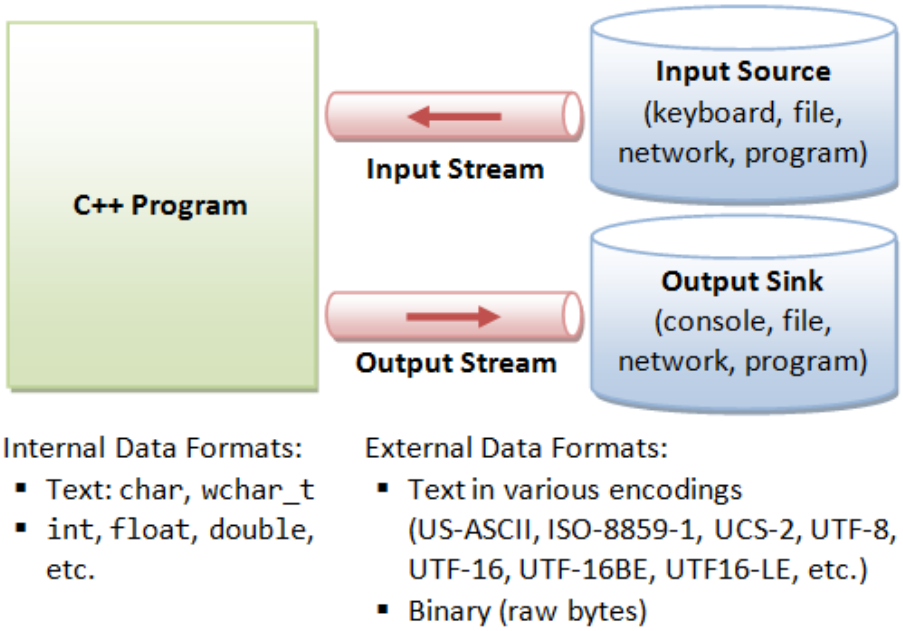
C++ Input and Output Streams

- To send data to and/or from an I/O device, a program must have I/O device read/write capabilities.
- Two methods to communicate to I/O devices in C/C++
  1. Writing directly to memory mapped I/O registers.
  2. Using the inbuilt C++ Stream I/O functionality.
- Streams are sequences of bytes
  - Also called a “flow of information”.
  - An input stream is an abstraction that “**produces**” information from the input device to the program.
  - An output stream is an abstraction that “**consumes**” information from the program to the output device.
  - A stream is linked to a physical device by the C++ I/O system.
  - On a disk drive, streams normally connect to files on the disk not to the disk as a whole.



C++ I/O Streams

- One of the great strengths of C++ is its I/O Streams Classes
- C++ Streams
  - A 'stream' is internally nothing but a series of characters flowing in and out of a program.
  - Input operations: data bytes flow from an input source (such as keyboard, file, network or another program) into the program.
  - Output operations: data bytes flow from the program to an output sink (such as console, file, network or another program).
- Streams as an intermediary
  - Streams acts as an intermediary between the programs and the I/O device.
  - They provide **abstraction**, so as to archive device independent I/O operations.
  - Therefore the programmer does not need to worry about low level I/O routines to extract or send data.
  - All I/O devices are effectively treated the same, using the same I/O interfaces functions and/or syntax.



Source: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/cp10_IO.html

C++ I/O Streams

- C++ streams provide a logical interface
  - By default, the C++ Standard streams are linked to the console, but they can be redirected to other devices or files in your program.
  - All streams have similar properties.

- Functionality provided in the C++ Standard I/O system `#include <iostream>`
  - Other header files may be needed for specific functionality
  - Operates in the `namespace std`
  - There is an older I/O system based on the header file `#include <iostream.h>` that is now obsolete and has less functionality.

- So far, we have been using the `<iostream>` standard library to provide **`cout <<`** and **`cin >>`** functionality for writing to standard output, and reading from standard input, respectively.

```
#include <iostream>  // C++ Standard I/O stream

using namespace std;

int main(int argc, char* argv[])
{
   int age = 0;
   char name[20] = {};

   // Display OUTPUT to screen
   cout << "Enter your name: ";

   // Request INPUT from Keyboard
   cin >> name;

   // Display OUTPUT to screen
   cout << "Enter your age: ";

   // Request INPUT from Keyboard
   cin >> age;

   // Display OUTPUT to screen
   cout << name << " is " << age << " years old";
   cout << endl << endl;

   return 0;
}
```

C++ I/O Streams

Predefined streams are automatically opened when a program starts if the `#include <iostream>` line is present:

- `cout` : Associated with standard output (normally the screen and is buffered)
- `cin` : Associated with standard input (normally the keyboard)
- `cerr` : Linked to standard output (not buffered)
  - Used to output error information
  - Non-buffered so output is written immediately
- `clog` : Linked to standard output (buffered)
  - Used to output information to be logged (e.g. for later analysis)
  - Buffered so output is written only when the buffer is full

- The predefined object **`cout`** is an instance of **`ostream`** class.
  - The stream **insertion operator** `<<` may be used more than once in a single statement.
  - The `<<` operator is overloaded to output data items of built-in types `int`, `float`, `char`, `double`, `strings` and pointer values.
  - `endl` or `'\n'` can be used to add a new-line.

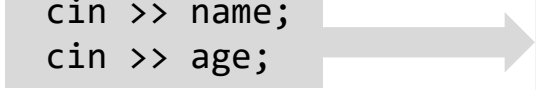
```
char str[] = "Hello World\n";
cout << "str = " << str << endl;

int num1 = 78;
float num2 = 3.854;

cout << num1 << " " << num2 << endl;
```

C++ I/O Streams

- The predefined object **`cin`** is an instance of **`istream`** class.
  - The stream **extraction operator** `>>` may be used more than once in a single statement.

    Equivalent statements:

    ```
    cin >> name;
    cin >> age;
    ```

    ```
    char name[20] = {};
    float age = 0;

    cin >> name >> age;

    cout << name << " is ";
    cout << age << " years old\n";
    ```

  - White space `"  "` or carriage return ↵ will cause `cin` operation to finish.

```
To get text with whitespaces use:   cin.getline(name,20);
```

- The predefined object **`cerr`** is an instance of **`ostream`** class.
  - `cerr` is **un-buffered** and each stream insertion to `cerr` causes its output to appear immediately on the console.

```
char str[] = "error messages";

cerr << "value = " << str << endl;
```

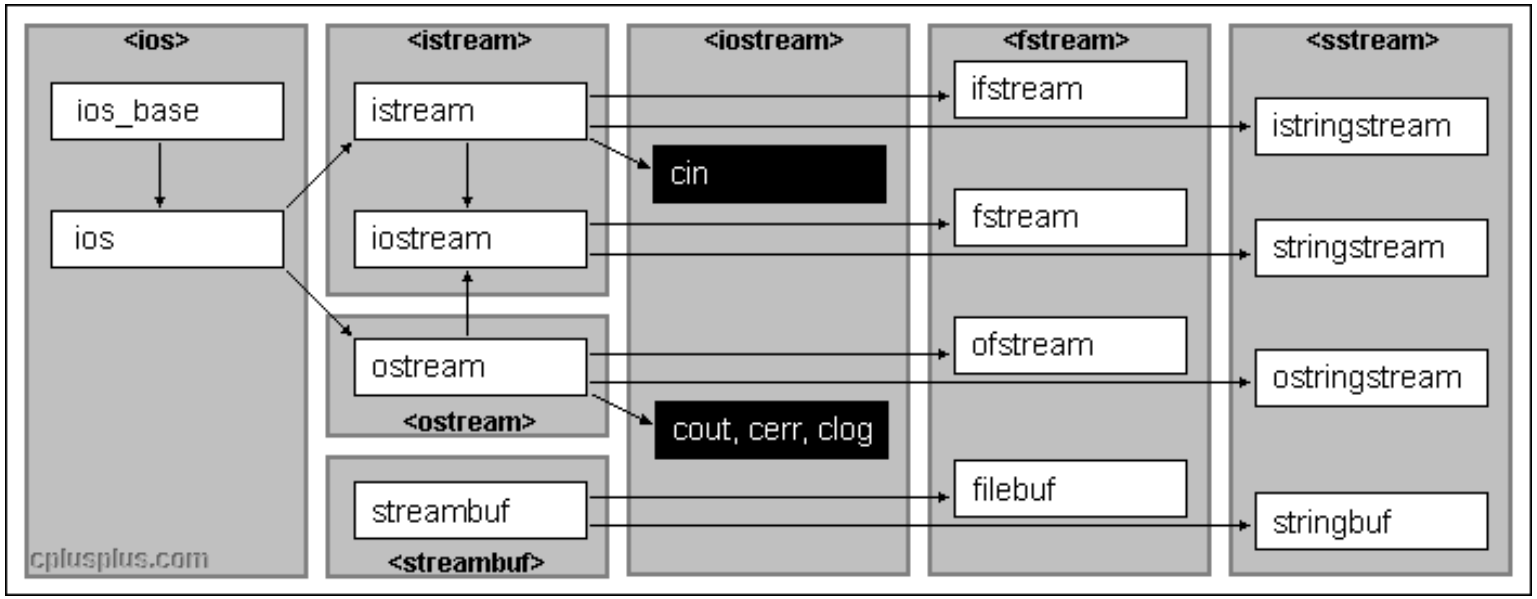
- The predefined object **`clog`** is an instance of **`ostream`** class.
  - clog is buffered, thus means that each insertion to clog could cause its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

```
char str[] = "debug information";

clog << "value = " << str << endl;
```

- *NOTE*: VS C++ also opens up 16-bit versions (`wchar_t`) of each stream to support languages such as Chinese: `wcin`, `wcout`, `wcerr` and `wclog`.

Input / Output Library Structure

- The `<iostream>` library is an object-oriented library that provides input and output functionality using streams.



- `<ios>` Base class for streams (type-dependent components).
- `<istream>` Header providing the standard input and combined input/output stream classes.
- `<ostream>` Header providing the standard output stream classes.
- `<streambuf>` Header providing the streambuf buffer class for input/output streams.
- `<iostream>` Declares the objects used for standard input and output (eg. `cin` & `cout`).
- `<fstream>` Defines the file stream classes to manipulate files using streams.
- `<sstream>` Defines classes to manipulate `string` objects as if they were streams.

Source: http://www.cplusplus.com/reference/iolibrary/

C++ I/O Stream Formatting

- So far we have been relying on the C++ I/O system default formats
  - but we can precisely control the format of data in the stream by either:
    - using member functions of the `ios` class (input/output stream)
      - E.g: `cout.setf(), cout.unsetf(), cout.width(), cout.precision(), cout.flush()`
    - Or by using manipulator functions using the insertion operator
      - E.g: `<< setprecision(3), << setw(20), << fill('#')`

Skip to slide 19

- Example using `ios_base` member function `setf()` with `showpos` and `scientific` flags.

```
using namespace std;

void main()
{
  // show default output
       cout<< 123 <<"\t"<< -456 <<"\t"<< 123.45 <<     endl;

  // turn on showpos and scientific flags
       cout.setf(ios::showpos);
       cout.setf(ios::scientific);

       cout<< 123 <<"\t"<< -456 <<"\t"<< 123.45 <<     endl;
}
```

Formatting with the "ios" stream format flags

`ios` class declares a bit enumeration called `fmtflags` which control the output of the stream object. The standard implementation of the `setf()` method is:

```
fmtflags setf (fmtflags flags); // returns previous flag setting
```

Field	Member Flag	Effect when set
independent flags *(optional flags)*	`boolalpha`	read/write bool elements as alphabetic strings (true and false)
	`showbase`	write integral values preceded by their corresponding numeric base prefix
	`showpoint`	write floating-point values including always the decimal point
	`showpos`	write non-negative numerical values preceded by a plus sign (+)
	`skipws`	skip leading whitespaces on certain input operations
	`unitbuf`	flush output after each inserting operation
	`uppercase`	write uppercase letters replacing lowercase letters in certain insertion operations
numerical base (`basefield`)	`dec`	read/write integral values using decimal base format
	`hex`	read/write integral values using hexadecimal base format
	`oct`	read/write integral values using octal base format
float format (`floatfield`)	`fixed`	write floating point values in fixed-point notation
	`scientific`	write floating-point values in scientific notation
adjustment (`adjustfield`)	`internal`	the output is padded to the *field width* by inserting *fill characters* at a specified internal point
	`left`	the output is padded to the *field width* appending *fill characters* at the end
	`right`	the output is padded to the *field width* by inserting *fill characters* at the beginning

Each group is Bit masked so only 1 flag in each group can be set

Formatting with the "ios" stream format flags

- The formatting flags are actually separate bits, so it is possible to set many bits (flags) at once using the bitwise OR operator

```
cout.setf (ios::scientific | ios::showpos); // saves a line of code
```

- Two forms of `std::ios_base::setf()`
  1. The first form is used to set **independent** format flags: These are the flags in the top half of the table in the previous slide. These flags have to be unset to deactivate them.
  2. The second form is used to set a value for one of the **selective flags**, These are the flags in the `adjustfield`, `basefield` and `floatfield` fields. These fields are bitmasked – when one flag is set in a field, the other flags are unset.

flag value	equivalent to
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

```
// activate showbase (first form)
cout.setf( ios::showbase );

// set hex as the basefield (second form)
cout.setf( ios::hex, ios::basefield );

cout << 100 << '\n';
```

Reference: http://en.cppreference.com/w/cpp/io/ios_base/fmtflags

Formatting with the "ios" member functions

- To unset (clear) the individual stream flags - use the unsetf() function

```
void unsetf(fmtflags flags);
```

  - Only the specified flags are cleared
  - For example: Clear the showbase flag

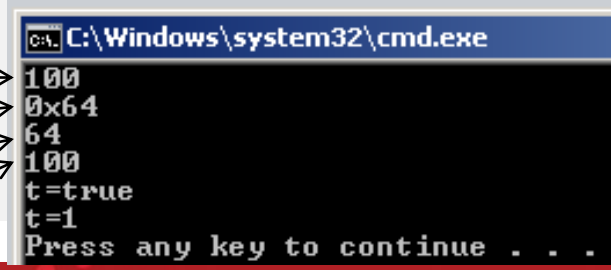
```
cout.unsetf(ios::showbase);
```

  - This performs a bitwise AND with the cout flags

- Example how to set and unset flags using `setf()` and `unsetf()`:

```
cout << 100 << '\n';                    // default output

cout.setf(ios::hex, ios::basefield );// set hex as the basefield
cout.setf(ios::showbase );              // ACTIVATE showbase
cout << 100 << '\n';

cout.unsetf (ios::showbase );           // DEACTIVATE showbase
cout << 100 << '\n';

cout.unsetf (ios::hex );                // go back to default (decimal)
cout << 100 << '\n';

bool t = true;
cout.setf(ios::boolalpha);
cout << "t=" << t << endl;
clog << "t=" << t << endl;
```

Default output
In hex with base shown
In HEX without base
unsetf() used to go back to default



Formatting with the "ios" member functions

- To obtain the current flag settings without making any changes, use:

```
ios::fmtflags f; // declare a variable of type fmtflags
f = cout.flags();
```

- This may be useful when attempting to determine the current state of the system as in this example:

Example of how to test current flag settings:

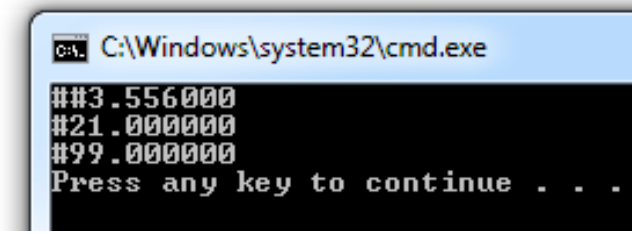
```
ios::fmtflags f;
f = cout.flags();           // Get current flag settings

if(f & ios::showpos)
   cout << "showpos is set for cout" << endl;
else
   cout << "showpos is cleared for cout" << endl;

cout.setf(ios::showpos);    // Setting showpos for cout
f = cout.flags();           // Get current flag settings

if(f & ios::showpos)
   cout << "showpos is set for cout" << endl;
else
   cout << "showpos is cleared for cout" << endl;

cout.unsetf(ios::showpos); // Unset showpos for cout
f = cout.flags();           // Get current flag settings

if(f & ios::showpos)
   cout << "showpos is set for cout" << endl;
else
   cout << "showpos is cleared for cout" << endl;
```

- The same format specifiers used for the cout stream can equally be used to control formatting text files written to disk.

Formatting with the "ios" member functions

- Member functions also exist to control the field `width`, `precision` and `fill` character:

```
streamsize width(streamsize w);// not persistent
streamsize precision(streamsize p);
char fill(char p);
```



  - `streamsize` is defined as a kind of integer (an `int` with limited range)

- When a value is output it uses a predefined space to display the value
  - By default a value uses just the space needed to display it
- Using `width()` allows the programmer to specify the space that each value uses
  - The previous value of width is returned by the function. Useful restore it later.
  - If the width specified is smaller than the space required for a value, the field is simply overrun - no truncation occurs.
  - If the field width specified is larger than the value being output, then that space is filled with the fill character (Default is whitespace).
  - The width setting only applies for the next insertion or extraction.

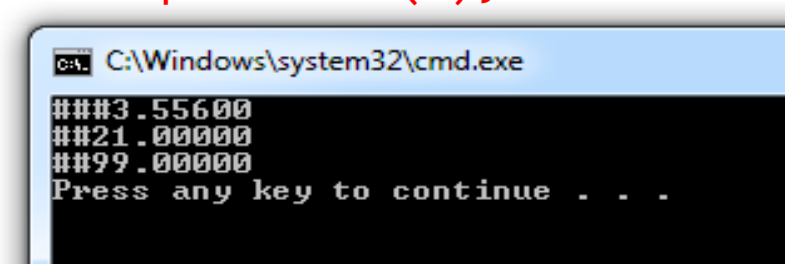
```
streamsize w = 10;

cout.fill('#'); //use non-whitespace
cout.width(w);  // not persistent

cout.setf(ios::fixed); // persistent

cout << (float)3.556 << endl;
cout.width(w);  // not persistent
cout << (float)21 << endl;
cout.width(w);  // not persistent
cout << 98.99999998 << endl;
```

Formatting with the "ios" member functions

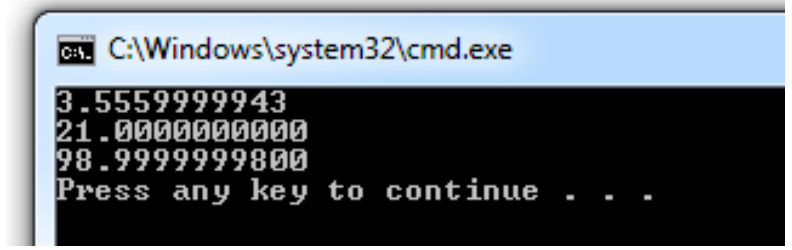
- The `precision()` member function sets the number of digits displayed after the decimal point for `float` or `double` type variables
  - The default is typically six (6) digits
  - If the default output is used (decimal), the precision function will only display the specified number of digits
  - The `float`ing-point (or `double`) number is rounded before being displayed
  - Trailing zeros are displayed when using scientific notation
  - Similar to the `width()` function, the `precision()` function returns the previous precision value.

```
streamsize w = 10;

cout.fill('#');
cout.width(w);
cout.precision(w);
cout.setf(ios::fixed);

cout << (float)3.556 << endl;
cout.width(w);
cout << (float)21 << endl;
cout.width(w);
cout << 98.99999998 << endl;
```

`cout.precision(5);`



`cout.precision(10);`



Formatting using the I/O Manipulators

- The second manner in which output can be formatted is using the I/O Manipulators which utilise the insertion(<<) and extraction(>>) operators.
  - A manipulator can be used as part of the I/O expression
  - The header file `#include <iomanip>` must be included to use the I/O manipulators
  - Using either the `ios` flags and member functions, or the I/O manipulators, the same results can be achieved.

Manipulator	Effect
`fixed`	Turns on fixed flag
`endl`	Inserts a newline and flushes the buffer
`setw(int w)`	Set the field width to w
`setprecision(int p)`	Set number of digits of precision to p
`scientific`	Output floating-point values using scientific notation
`setfill(int ch)`	Sets fill character for padding to ch
`showbase`	Generates a prefix indicating numeric base of an integer

Example notation:

```
cout << setw(10) << fixed << (float)3.142 << endl;
cout << hex << 100 << endl;
```

Reference: http://www.cplusplus.com/reference/iomanip/

C++ I/O Stream Formatting

Back to our problem. How to *nicely* format array output?

```
double myarray[3][3] = { {3.25 , 3.1    , 52.0001     },
                         {5.2  , 6.8892 , 6.225       },
                         {1.0  , 885.2  , 52.554444444}  };
```

Two ways to output data in columns with a precision of 3 decimal places:

Using `ios` class members:

```
// other code ...

for(int i=0; i<3; i++)   // rows
{
  for(int j=0; j<3; j++) // column
  {

        //  Missing code…

    cout << myarray[i][j] << "\t";
  }
  cout << "\n";
}
```

Using I/O Manipulators:

```
#include <iomanip>
// other code ...

for(int i=0; i<3; i++)   // rows
{
  for(int j=0; j<3; j++) // column
  {

        //  Missing code…

    cout << myarray[i][j] << "\t";
  }
  cout << endl;
}
```

C++ File I/O

- C++ can read / write either binary or text files.
  - Text files have input / output translation
    - Text files are human readable.
  - Binary files have no input / output translation
    - Binary files are often not human readable and may be compiled executable, compiled libraries, or data in binary format.

- To perform I/O on text or binary files, `#include <fstream>`
  - `<fstream>` defines a number of important classes and values
  - Using the functions within these libraries greatly simplifies access to I/O storage devices.
  - The programmer does not have know the details of how to open / save the file on a disk drive.
    - The operating system takes care of the file allocation system, whether it be Windows (NTFS, FAT), iOS, UNIX/Linux/Android or other operating system. The same source code can be used for multiple operating systems.

```
ifstream
fstream
ofstream
filebuf
```

Note: In this section the word "*file*" specifically refers to text or binary files usually found on a disk drive

Opening a File

- There are three stream class types associated with file manipulation
  - class `ifstream` will allow you to open file as an input stream ie. READ
  - class `ofstream` will allow you to open (or create) a file as an output stream. ie. WRITE
  - class `fstream` will allow you to open (or create) a file as either an input or output stream. ie. Both READ/WRITE

- The open() function associates (links) a file with a stream
  - open() is a member of all three stream classes
  - The function prototypes for `ifstream` and `ofstream` are very similar, but below is the form for the `fstream` class:

```
void fstream::open(const char * filename,
                   ios::openmode mode = ios::in | ios::out)
```

- The function prototype holds the key to correctly handling the file
  - `filename` is the name of the file which can contain a path specifier, e.g: `c:\cpp\test.cpp`
  - `mode` determines how the file is opened which must be one the specifiers in the next table.

File Writing Modes

To write a stream to a file:

- create a stream object
- open it
- check it opened
- write to it
- and close the file

```
ofstream myStream;  // make a new ofstream object
myStream.open("test.txt", ios::out | ios::app);
if(!myStream)       // make sure it opened
  myStream << "Add this line to the file" << endl;
myStream.close();   // close file
```

Mode	Description
ios::app	All output to the file is appended.
ios::ate	Causes a seek to the end of the file i.e. the current location will be at the end of the file when opened. I/O operations can however still occur anywhere within the file.
ios::binary	Opens file in binary mode. Data sent to the stream is what is exactly written to the file. No text character or substitution takes place at any stage.
ios::in	File is capable of input.
ios::out	File is capable of output.
ios::trunc	Causes any existing file of the same name to be destroyed and the new file to be zero length. (overwrite any existing files of the same name)

Two or more of these values can be combined by OR-ing them together.

Opening a File

- By default all files are opened in text mode
  - which enables character translations such as the newline character (`'\n'`) which will often be translated into a carriage return-line feed sequence
  - When creating an output stream using `ofstream` any existing file of that name is automatically truncated (ie. Overwritten)
- File can be opened in two different ways (they are effectively the same):

```
ofstream mystream;              // create output file stream object
mystream.open("test.txt");      // open test.txt for output
```

```
ofstream mystream("test.txt");// create & open test.txt for output
```

- Most compilers do not need the mode set, as this is apparent from the use of `ofstream`. However, some compilers require the programmer to specify `"in|out"` when using `fstream`
- The programmer should check if the operation was successful
  - First way: check if the fstream object is `!open`
  - Second way: check using the inbuilt `is_open()` function

Opening and Closing a File

- If the fstream fails to open, the stream will then evaluate to false when tested (either method works):

```
myStream.open("test.txt");
if(!myStream)
{
  cerr << "Cannot open file";
  // Handle the error
}
```

```
myStream.open("test.txt");
if(!myStream.is_open())
{
  cerr << "File is not open";
  // Handle the error
}
```

  - The `is_open()` function is well suited for times when the file should be open but the programmer is unsure whether it is still open
  - `is_open()` is a member of `fstream`, `ifstream` and `ofstream`
  - `is_open()` will return true if the stream is linked to an open file, else the result is false

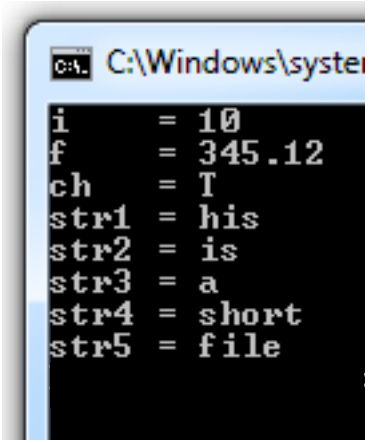
- Files should not be left open
  - To close a file (after input and/or output is complete) call the member function `close()`

```
myStream.open("test.txt");
myStream.close(); // do this ASAP
```

Reading from a Text File

- The simplest way to read / write to a text file is to use the >> (extraction) or << (insertion) operators
  - `extraction` >> is used to read from a file
  - `insertion` << is used to write to a file
- Example: Reading from a text file
  - When using the >> operator to read a text file there are a few important points:
    - Character translation can occur
    - Leading whitespaces are extracted and discarded
    - Any read operation using the >> operator stops as soon as the first whitespace is encountered

```
10   345.12
This is a short file
```




```
int i;
float f;
char ch;
char str1[80], str2[80], str3[80];
char str4[80], str5[80];

ifstream infile ("test1.txt");
if(!infile)
{
   cerr << "Error!" <<  endl;
   return 0;
}

infile >> i;    // read integer
infile >> f;    // read float
infile >> ch;   // read char
infile >> str1; // read string
infile >> str2; // read string
infile >> str3; // read string
infile >> str4; // read string
infile >> str5; // read string

infile.close();

cout << "i    = " << i    << endl;
cout << ”f    = " << f    << endl;
```

Reading and Writing Binary Files

- Not all files are text files
- Binary files let the programmer store data without modification and are more efficient (faster and smaller)
  - When working with binary files the programmer must open it using the `ios::binary` mode specifier
- There are two main methods used to read or write a binary file
  1. Use member functions such as `get()` and `put()`
     - By default these functions deal with one byte at a time
  2. Use block I/O functions such as `read()` and `write()`
     - These functions deal with whole blocks of data
     - The programmer can specify how much data is accessed when using the block functions

```
02 A4 78 09 ...
54 A5 D8 E9 ...
0A B4 7E 09 ...
F2 AC 7F 0D ...
```

Writing One Character at a Time – put()

- Multiple versions of the `put()` functions exist, however they follow a similar form to the one used by Visual Studio.
- To **write** a binary word to a file use the `put()` function:

```
ostream &put(char ch);
```

  - This function writes one character (byte) to the stream and returns a reference to the stream.
  - This is useful if we want to write a single 8-bit word to a file, like 0x01, 0x02 or 0xF4
  - You can also write ASCII chars arrays or strings (see example file).
- Remember to include the header file `#include <fstream>` and the `ios::binary` flag.

Example: Writing binary file using put() :

```
int main()
{
  char line[80] = {0x30,0x31,0x32,0x41,0x42};

  ofstream outfile ("test-binary.hex",
                    ios::out | ios::binary);

  if(!outfile)
  {
     cerr << "Cannot create file\n";
     return 0;
  }

  int i = 0;
  while(line[i])  // not NULL char
  {
     outfile.put(line[i]);
     i++;
  }

  outfile.close();
  return 0;
}
```

Reading One Character at a Time – get()

- Multiple versions of the `get()` functions exist too.
- To **read** a binary word from a file use the `get()` function:

```
istream &get(char &ch);
```

  - This function reads a single character (byte) at a time and places the value in the address passed in by reference in the argument parameter, `ch`.
  - The return value is a reference to the stream. This reference value is null if the end of the file is reached.

Example: Reading a binary file using get() :

```
int main()
{
  char ch;

  ifstream infile ("test-binary.hex",
                          ios::in| ios::binary);

  if(!infile)
  {
     cerr << "Cannot open file\n";
     return 0;
  }


  while(infile.get(ch))  // not NULL char
  {
     cout << ch << endl; // display char
  }
  infile.close();

  cout << endl;
  return 0;
}
```

Writing and Reading Blocks of Data

- Rather than read just a single character from an file, C++ allows the programmer to read and write large chunks
- To read and write blocks of binary data, use `read()` and `write()` member functions of `fstream` class.
- The prototype for `read()` is:

```
istream &read(char *buf, streamsize num);
```

  - The `read()` function reads `num` bytes from the associated stream and places them in the buffer pointed to by `buf`

- Likewise, the prototype for `write()` is:

```
ostream &write(const char *buf, streamsize num);
```

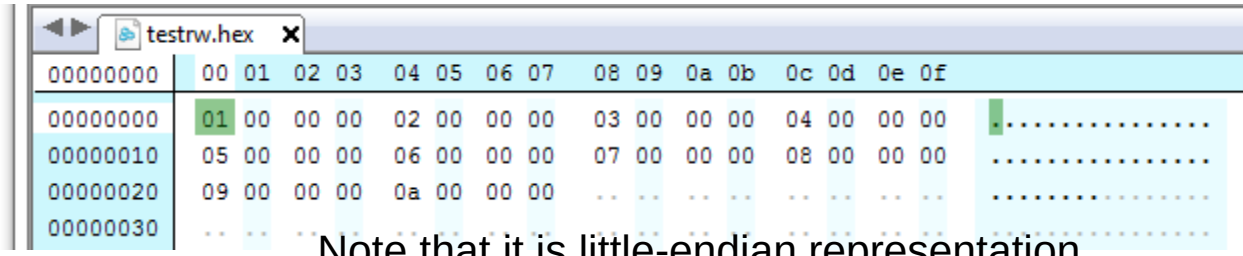
  - The `write()` function writes `num` bytes to the associated stream from the buffer pointed to by `buf`
  - The `streamsize` type is defined as a form of integer by the compiler. It is capable of holding the largest number of bytes that can be transferred in any one I/O operation.

Writing a Block of Data

```
int main()
{
  int dataOUT[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  ofstream outfile("testrw.hex", ios::out | ios::binary);

  if(!outfile)
  {
    cerr << "Cannot create file testrw.hex" << endl;
    return 0;
  }

  outfile.write ( (char*) &dataOUT, sizeof(dataOUT) );
  outfile.close();

  return 0;
}
```

- In this example the array to write is not of a `char` data-type. It's of type `int`.
  - Hence the array is type cast into a character type (`char`*) to fit the prototype requirements.
  - The `sizeof()` function is used here to determine how big the block to write needs to be.

- Because we are writing integers, 4 bytes will be written for every value. (i.e. 32 bit value)



Note that it is little-endian representation

Reading a Block of Data

```
int main()
{
  int data_IN[10] = {};  // double the size of what we need

  ifstream infile("testrw.hex", ios::in | ios::binary);
  if(!infile)
  {
    cout << "Cannot open file testrw.hex" << endl;
    return 0;
  }
  infile.read ( (char*) &data_IN, sizeof(data_IN) );
  infile.close();

  for(int i=0; i<(sizeof(data_IN)/sizeof(int)); i++)
    cout << data_IN[i] << "\t"; // display to screen

  cout << "Number of bytes read in = " << infile.gcount() << '\n';
  return 0;
}
```

- How does the programmer detect the End-Of-File (EOF) condition?
  - When reading from the file, `num` bytes are read. If the end-of-file is reached before `num` bytes have been read, the `read()` function simply stops reading.
  - You can find out how many bytes were read using the gcount() function.
  - Remember when reading from a file that the receiving buffer must be of at least the required length otherwise the array bounds will be exceeded.
  - The result from exceeding array bounds is unpredictable and can lead to the program crashing the host operating system.

More I/O Functions

- When working with text files a more efficient way to read a complete line from the file is:

```
istream getline(char *buf, streamsize num);
```

  - `getline()` reads `num` characters into an array pointed to by `buf` until either:
    - `num-1` characters have been read
    - Newline character is found (`'\n'`)
    - End-Of-File (`EOF`) reached
  - If the newline character (`'\n'`) is found in the input stream it is extracted but not put into `buf`

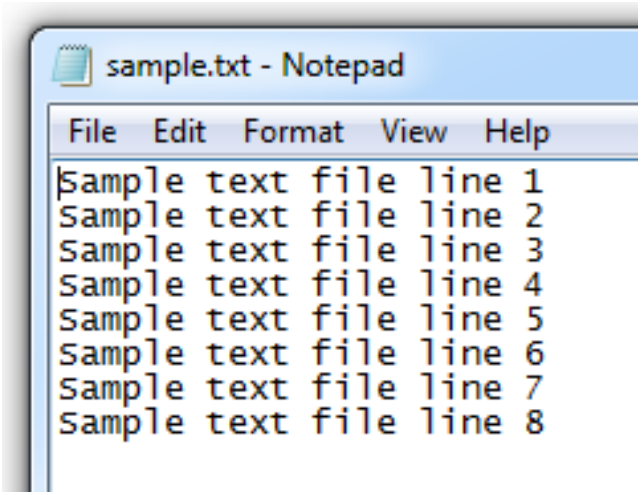
- Another useful `getline()` overloaded version is:

```
istream getline(char *buf, streamsize num, char delim);
```

  - Works the same as above except the `delim` character can be used instead of newline
  - Can be extremely useful for processing *.csv files, etc.

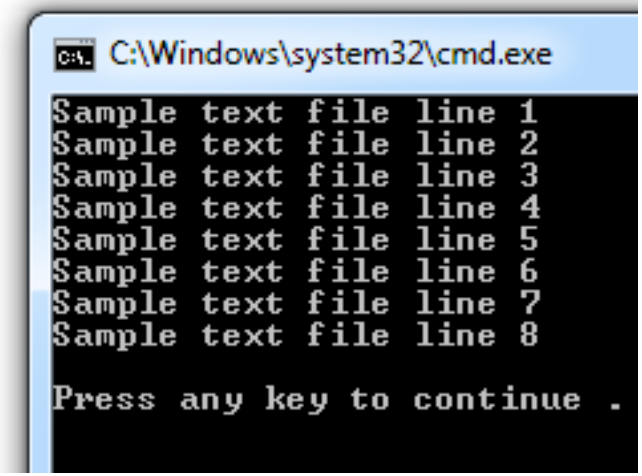
getline() Example

Sample text file with 8 line:




```
int main()
{
  char buff[256] = {};

  ifstream infile("sample.txt", ios::in | ios::binary);
  if(!infile)
  {
    cout << "Cannot open file sample.txt" << endl;
    return 0;
  }

  while(!infile.eof()) // runs until end of file
  {
     infile.getline(buff, sizeof(buff));
     cout << buff << endl;
  }

  infile.close();

  return 0;
}
```

Output:



Try the above code with a whitespace ' ' as a delimiter !!!
How does the output change? What advantages does it have?

Strings

- Up until now we have been explicitly using C-strings (ie. char arrays):

```
char myStr[] = "hello world";
```

- Strings are objects that represent sequences of characters, just like above however they far more powerful as they come with in-built functionality.

```
string myStr = "hello world"; // declare and define a string object

cout << myStr << endl;        // cout deals with them just like C-strings

myStr = myStr + " this is new text";    // add text to the string
cout << myStr << endl;

int size = myStr.length();              // uses its length member function
cout << "myStr size = " << size << endl;

string Str1 = "Strings can be ";
string Str2 = "Added together!";
string Str3 = Str1 + Str2;              // Str1 += Str2 would also work
cout << Str3 << endl;
```

- Strings are not a built-in type, like an int or float. They are part of the C++ string Class, which is part of the C++ Class Library. The string class provides an object-oriented approach to string handling and has many useful features.

Strings

- Declaring a string is easy:

```
using namespace std;

string my_string;
```

Or

```
std::string my_string;
```

- String Comparisons (big advantage of char *)

```
#include <string>
string passwd;

getline(cin, passwd, '\n');
if(passwd == "xyzzy") {
    cout<< "Access allowed";
}
```

No special comparison function required! No `for(;;)` loops…

- Can be accessed in the same way a `char[]` is:

```
string my_string("Hello world");

for(int i = 0; i < my_string.length(); i++)
  cout << my_string[i]; // 1 char at a time
```

Strings

Modifying Strings:

– Erasing pats of string:

```
string my_removal = "remove aaa";
my_removal.erase(7, 3); // erases aaa
```

```
//Swap example:

string s1( "abc" );
string s2( "def" );
s1.swap( s2 );

// now s1 = "def",
// and s2 = "abc" now
```

– Splicing 2 strings together:

```
string my_string = "ade";
my_string.insert(1, "bc"); // my_string is now "abcde"
```

Retrieving a c-style string (char*):

```
string my_string = "1050";
int number = 0;

number = atoi(my_string.c_str());  // convert value to integer
cout << number << endl;
```

For full list of operators, visit: http://www.cplusplus.com/reference/string/string/

Strings

Searching:

The string class supports simple searching and substring retrieval using member functions: `find()`, and `rfind()`.

```
int find(string pattern, int position);
```

A simple string search example:

```
string myStr = "cat, dog, cat, dog, horse, lion";
int cat_appearances = 0;

for(int i = myStr.find("cat", 0); i < myStr.length(); i = myStr.find("cat", i))
{
    cat_appearances++;
    i++;  // Move past the last discovered instance to avoid finding same string
}
cout << "number of instances of cat = " << cat_appearances << endl;
```

`rfind()` works in almost the same way, except that searching begin at the very end of the string, rather than the beginning.

Helpful website: http://www.cprogramming.com/tutorial/string.html