# CS 103 Unit 7 Slides

File I/O

Mark Redekopp

# Get the Files for Today

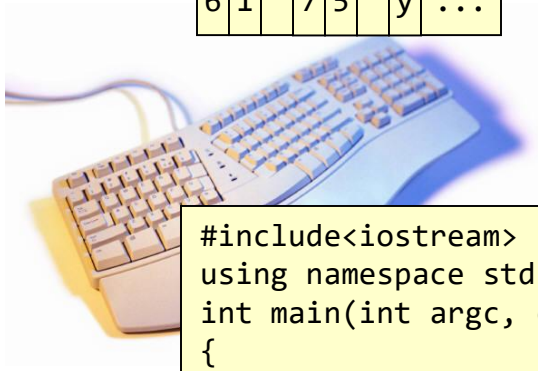
- Go to your cs103/examples directory
  - $ wget http://ee.usc.edu/~redekopp/cs103/redir_pipe.tar
  - $ tar xvf redir_pipe.tar
  - $ wget http://ee.usc.edu/~redekopp/cs103/cinfail.cpp
  - $ wget http://ee.usc.edu/~redekopp/cs103/file_io_ex.tar
  - $ tar xvf file_io_ex.tar

# I/O Streams

- I/O is placed in temporary buffers/streams by the OS & C++ libraries
- cin pulls data from an input stream known as 'stdin' (standard input)
  - It is usually the stream coming from the keyboard
- cout puts data in an output stream known as 'stdout' (standard output)
  - It is usually directed to the monitor

input stream [stdin] (user types all at once):

| 6 | 1 |  | 7 | 5 |  | y | ... |
|---|---|---|---|---|---|---|---|



```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  int dummy, x;
  cin >> dummy >> x;

}
```

stdin

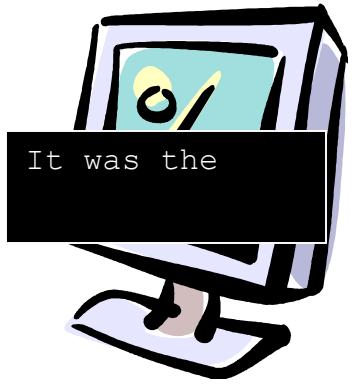
|  | y | ... |
|---|---|---|

```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  cout << "It was the" << endl;
  cout << 4;
}
```

output stream (stdout) in OS:

stdout

| I | t |  | w | a | s |  | t | h | e | \n | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|

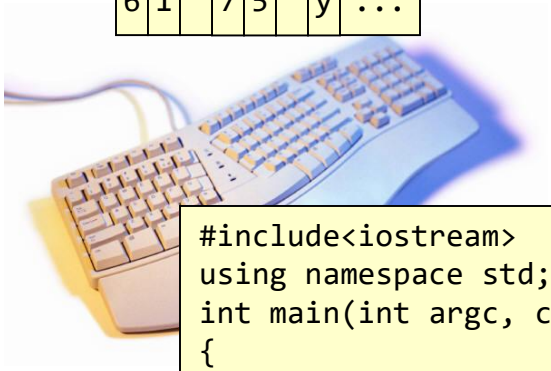


# I/O Streams

- '>>' operator used to read data from an input stream
  - Always stops at whitespace
- '<<' operator used to write data to an output stream
  - 'endl' forces a flush…Flush forces the OS to move data from the internal OS stream to the actual output device (like the monitor)

input stream (user types all at once):

| 6 | 1 |  | 7 | 5 |  | y | ... |
|---|---|---|---|---|---|---|---|



```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  int dummy, x;
  cin >> dummy >> x;

}
```

input stream:

|  | y | ... |
|---|---|---|

```
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{
  cout << "It was the" << endl;
  cout << 4;
}
```

output stream in OS:

| I | t |  | w | a | s |  | t | h | e | \n | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|


output stream after flush:

| 4 |
|---|

# File I/O Intro

- What methods does a user have to provide a program input:
  - cin
  - Command line (argc, argv)
- Now a new method: File I/O (accessing data in files)
- Primary method for a program to read/write files:
  - File streams [Main subject of our lecture]
- OS-based tools (for scripts) to read/write file data
  - I/O Redirection via the OS (use of '<' and '>' at command line)
  - Pipes via the OS (use of | at command line)

# Redirection & Pipes

- The OS (Linux or Windows or Mac) provides the following abilities at the command line
- '<' redirect contents of a file as input (stdin) to program
  - ./simulation < input.txt
  - OS places contents of input.txt into 'stdin' input stream which broke can access via 'cin'
- '>' redirect program output to a file
  - ./ simulation < input.txt > results.txt
  - OS takes output from 'stdout' produced by cout and writes them into a new output file on the hard drive: results.txt
- 'l' pipe output of first program to second
  - stdout of first program is then used as stdin of next program

# Redirection & Pipe Examples

- $ ./lab5_sol < input.txt
  - Redirects contents of input.txt to stdin (i.e. cin) in lab5 program
- Get the demo files:
  - Go to your cs103/examples directory
  - $ wget http://ee.usc.edu/~redekopp/cs103/redir_pipe.tar
  - $ tar xvf redir_pipe.tar
  - $ make randgen
  - $ make average
- Run them without using redirection and pipes
  - $ ./randgen 20 10
    - Notice 20 values between 1-10 are output on stdout/cout
  - $ ./average
    - Now type in a list of numbers followed by typing Ctrl-D

```
0 10 10 100 50
0 200 220 20 30
1 80 180 25 25
1 180 50 30 60
2
```

input.txt

# Redirection & Pipe Examples

- Output Redirection: >
  - $ ./randgen 20 10 > out.txt
  - Now inspect out.txt contents
  - What would have displayed on the screen is now in out.txt
- Input redirection: <
  - $ ./average < out.txt
  - The output captured from randgen is now used as input to average
- Pipes: |
  - $ ./randgen 20 10 | ./average
  - The output of randgen is fed as input to average

Other capabilities you can use for streams

# MORE ABOUT STREAMS

# Input & Output Streams

- There are other types of input and output streams other than cin and cout
- File streams gives the same capabilities of cin and cout except data is read/written from/to a file on the hard drive
  - Everything you do with cin using the '>>' operator you can now use to access data from a file rather than the keyboard
  - Everything you do with cout using the '<<' operator you can now use to output data to a file
- Let's learn more about streams '>>'…we'll see it in the context of cin and cout but realize it will apply to other streams we'll learn about next

# getline() and Lines of Text

- cin stops reading at whitespace
  - If you want to read a whole line of text use `cin.getline()`
    - It will read spaces and tabs but STOP at '\n'
  - `cin.getline(char *buffer, int max_chars)`
    - Reads `max_chars-1` leaving space for the null character

```
#include <iostream>
using namespace std;

int main ()
{
  char mytext[80];
  cout << "Enter your full name" << endl;
  cin.getline(mytext, 80);

  int last=0;
  for(int i=0; i<80; i++){
    if(mytext[i] == ' '){
       last = i+1;
       break;
    }
  }
  cout << "Last name starts at index: ";
  cout << last << endl;
  return 0;
 }
```

# Sample Code

- Get the sample code
  - $ wget http://ee.usc.edu/~redekopp/cs103/cinfail.cpp

# Input Stream Error Checking

- We can check errors when cin receives unexpected data that can't be converted to the given type
- Use the function cin.fail() which returns true if anything went wrong opening or reading data in from the file (will continue to return true from then on until you perform cin.clear())
- Try this code yourself and see what happens with and with out the check using fail()

```
#include <iostream>
using namespace std;

int main ()

{

  int x;
  cout << "Enter an int: " << endl;
  cin >> x;  // What if the user enters:
             //    "abc"


  // Check if we successfully read an int
  if(  cin.fail() ) {
    cout << "Error:  I said enter an int!";
    cout << " Now I must exit!" << endl;
    return 1;
  }

  cout << "You did it!  You entered an int";
  cout << " with value: " << x;

  return 0;
 }
```

# Understanding Input Streams

- **User enters value "512" at 1st prompt, enters "123" at 2nd prompt**

***int x=0;***

**X =** `0` **cin =** ` `

***cout << "Enter X: ";***

**X =** `0` **cin =** `5` `1` `2` `\n`

***cin >> x;***

**X =** `512` **cin =** `\n`

**cin.fail() is false**

***int y = 0;***

**Y =** `0` **cin =** `\n`

***cout << "Enter Y: ";***

**Y =** `0` **cin =** `\n` `1` `2` `3` `\n`

***cin >> y;***

**Y =** `123` **cin =** `\n`

**cin.fail() is false**

# Understanding Input Streams

- **User enters value "23abc" at 1st prompt, 2nd prompt fails**

***int x=0;***

X = 0 cin =

***cout << "Enter X: ";***

X = 0 cin = 2 3 a b c \n

***cin >> x;***

X = 23 cin = a b c \n

**cin.fail() is false**

***int y = 0;***

Y = 0 cin = a b c \n

***cout << "Enter Y: ";***

Y = 0 cin = a b c \n

***cin >> y;***

Y = xx cin = a b c \n

**cin.fail() is true**

# Understanding Input Streams

- **User enters value "23 99" at 1st prompt, 2nd prompt skipped**

***int x=0;***

**X =** `0` **cin =**

***cout << "Enter X: ";***

**X =** `0` **cin =** `2` `3` ` ` `9` `9` `\n`

***cin >> x;***

**X =** `23` **cin =** ` ` `9` `9` `\n`

**cin.fail() is false**

***int y = 0;***

**Y =** `0` **cin =** ` ` `9` `9` `\n`

***cout << "Enter Y: ";***

**Y =** `0` **cin =** ` ` `9` `9` `\n`

***cin >> y;***

**Y =** `99` **cin =** `\n`

**cin.fail() is false**

# Understanding Input Streams

- **User enters value "23 99" at 1st prompt, everything read as string**

*char x[80];*

*cout << "Enter X: ";*

*cin.getline(x, 80);*

X = [ ] cin = [ ]

X = [ ] cin = [ 2 | 3 | ␣ | 9 | 9 | \n ]

X = [ 23 99 ] cin = [ ]

**cin.fail() is false**

**NOTE: \n character is discarded!**

# More on Error Checking

- Use the fail() function to detect errors when attempting to read data
- If a call to fail() returns true then subsequent calls to fail() will continue to return true until you call clear()
- Use ignore() to clean out any text still in the input stream
- Try this code yourself and see what happens with and with out the check using fail()

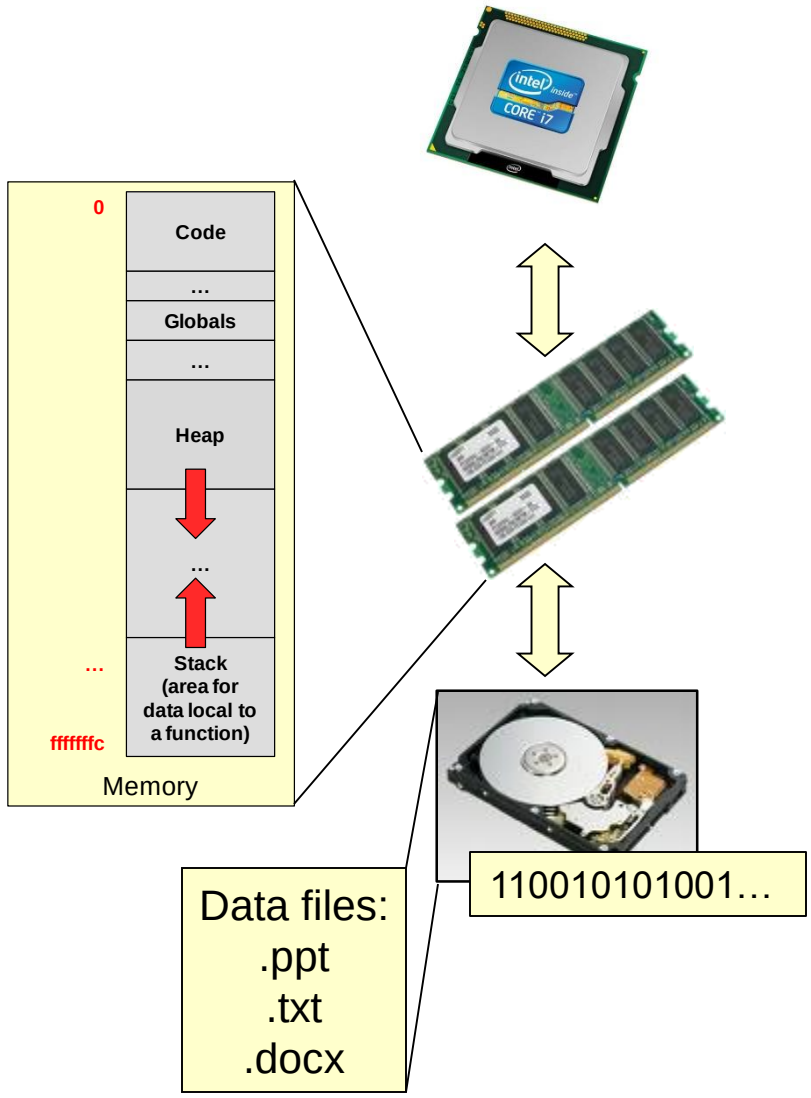
```
#include <iostream>
using namespace std;

int main ()

{

  int x;
  cout << "Enter an int: " << endl;
  cin >> x;  // What if the user enters:
             //    "abc"


  // Check if we successfully read an int
  while(  cin.fail() ) {
    cin.clear(); // turn off fail flag
    cin.ignore(256, '\n'); // clear inputs
    cout << "I said enter an int: ";
    cin >> x;

  }

  cout << "You did it!  You entered an int";
  cout << " with value: " << x;

  return 1;
 }
```

How your program can directly access data in files

# FILE STREAMS

# Computer Organization

- Processor can only talk directly to RAM
  - It needs "translation" to access data on the hard drive or other disk
- All code and data resides in RAM
  - All variables accessible in your program
- How do we access files
  - The OS provides routines to perform the translation



# Starting File I/O

- Just like Microsoft Word or any other application that uses files you have two options…
  - Read info from the file (like 'Open' command)
    - Use an 'ifstream' object to open the file
    - Read data from the file
    - Close it when you're done
  - Write info to the file (like 'Save As' command)
    - Use an 'ofstream' object
    - Write the data to a file
    - Close it when you're done

# Important Fact

- For your program to operate on data in a file…
- …you must read it into a C variable
- Everything we will see subsequently is simply how to get data into a variable
  - After that we can just process it normally
  - If we want to produce an output file we will just writing the variable values to the file using some more techniques
  - C/C++ provides functions that do the reading/writing for you

# Two Kinds of Files: Binary and Text

- We conceive of files as "streams" (linear arrays) of data
- Files are broken into two types based on how they represent the given information:
  - Text files: File is just a large sequence of ASCII characters (every piece of data is just a byte)
  - Binary files: Data in the file is just bits that can be retrieved in different size chunks (4-byte int, 8-byte double, etc.)
- Example: Store the number 172 in a file:
  - Text: Would store 3 ASCII char's '1','7','2' (ASCII 0x31,0x37,0x32) requiring 3 bytes
  - Binary: If 172 was in a 'char' var., we could store a 1-byte value representing 172 in unsigned binary (0xAC) or if 172 was in an 'int' var. we could store 4-bytes with value 0x000000AC

In this class we will only focus on Text files

# TEXT FILE I/O

# Activity

- Get the test files
  - $ wget http://ee.usc.edu/~redekopp/cs103/file_io_ex.tar
  - $ tar xvf file_io_ex.tar
- sum_from_file_exercise

# Text File I/O

- Use **ifstream** object/variable for reading a file
  - Can do anything 'cin' can do
- Use **ofstream** object/variable for writing a file
  - Can do anything 'cout' can do
- Must include `<fstream>`
- Use '<<' and '>>' operators on the stream but realize operations are happening on data form the file

input.txt

```
5 -3.5
```

output.txt

```
Int from file is 5
Double from file is -3.5
```

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()

{

  int x; double y;

  ifstream ifile ("input.txt");

  if(  ifile.fail() ){  // able to open file?
    cout << "Couldn't open file" << endl;
    return 1;
  }

  ifile >> x >> y;
  if (  ifile.fail() ){
    cout << "Didn't enter an int and double";
    return 1;
  }

  ofstream ofile("output.txt");

  ofile << "Int from file is " << x << endl;
  ofile << "Double from file is " << y << endl;

  ifile.close();
  ofile.close();

  return 0;
 }
```

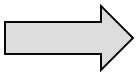
# Getting Lines of Text

- Using the >> operator to get an input string of text (char * or char [] variable passed to cin) ***implicitly stops at the first whitespace***
- How can we get a whole line of text (including spaces)
  - cin.getline(char *buf, int bufsize);
  - ifile.getline(char *buf, int bufsize);
  - Reads max of bufsize-1 characters (including newline)
- This program reads all the lines of text from a file

```
#include <iostream>
#include <fstream>
using namespace std;

int main ()

{

  char myline[100]; int i = 1;

  ifstream ifile ("input.txt");

  if(  ifile.fail() ){  // can't open?
    return 1;
  }

  ifile.getline(myline, 100);
  while ( ! ifile.fail()) {
   cout << i++ << ": " << myline << endl;
   ifile.getline(myline, 100);
  }

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

input.txt

```
The fox jumped over the log.\n
The bear ate some honey.\n
The CS student solved a hard problem.\n
```



```
1: The fox jumped over the log.
2: The bear ate some honey.
3: The CS student solved a hard problem.
```

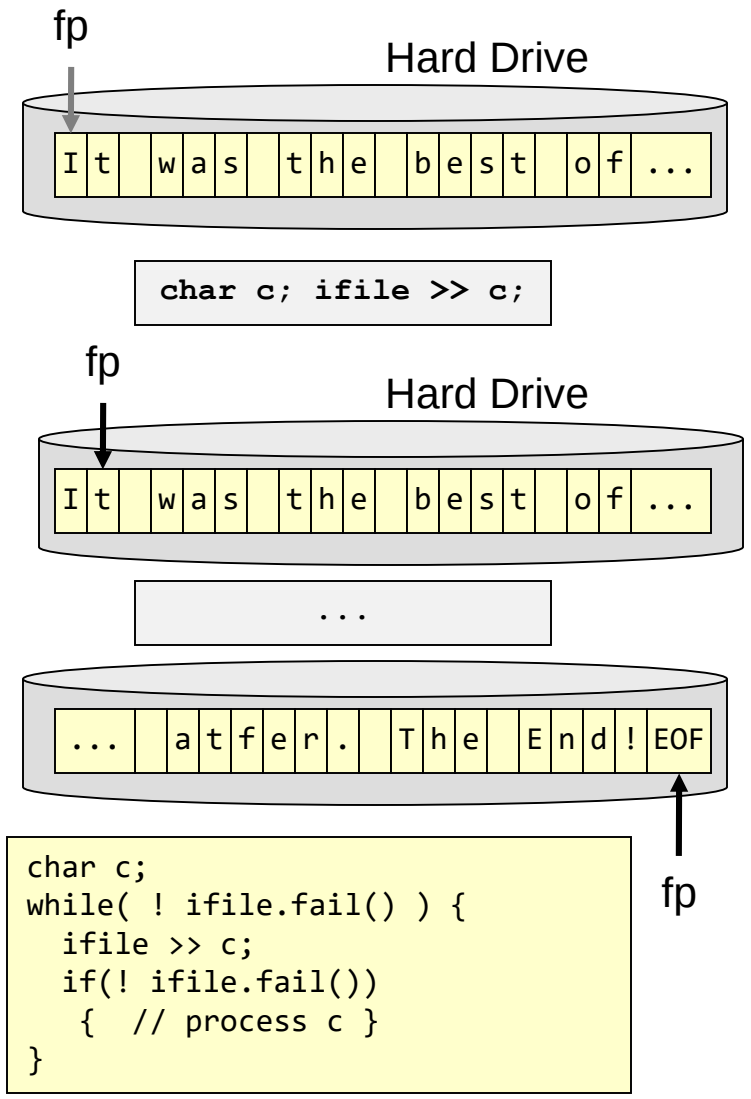
# Activity

- reverse_it exercise
- search_and_count exercise

# FILE LOCATION/POINTERS & INPUT OPERATORS

# File Access

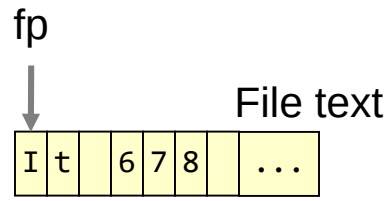
- Your ifstream object (ifile) implicitly keeps track of where you are in the file
- EOF (end-of-file) or other error means no more data can be read. Use the fail() function to ensure the file is okay for reading/writing



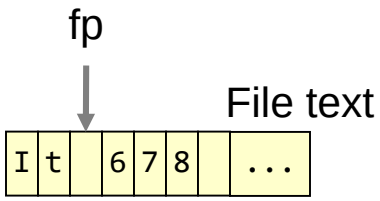
# >> Operator

- Recall that with cin the >> operator stops getting a value when it encounters whitespace and also skips whitespace to get to the next value
  - So do ifstream objects
- In the example on this slide, the spaces will NOT be read in
  - They will be skipped by the >> operator
- To get raw data from the file (including whitespaces) use the get() function

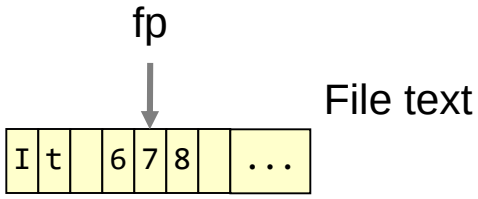
```
ifstream ifile("data.txt");
```




```
char s[40]; ifile >> s;
// returns "It" and stops at space
```



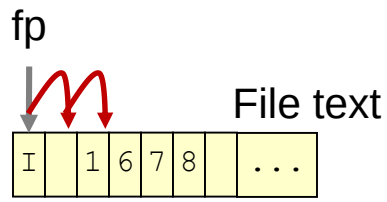

```
char x; ifile >> x;
// skips space & gives x='6'
```



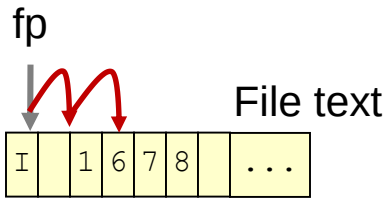
# >> Operator

- To get raw data from the file (including whitespaces) use the `ifstream::get()` function
  - Returns the character at the 'fp' and moves 'fp' on by one
- To see what the next character is without moving the 'fp' pointer on to the next character, use `ifstream::peek()` function
  - Returns the character at the 'fp' but does NOT move 'fp' on

```
ifstream ifile("data.txt");
```




```
char c = ifile.get(); // returns 'I'
C = ifile.get(); // returns ' '
```




```
char c;
ifile >> c; // returns 'I'
ifile >> c; // skips space and
            // returns '1'
```

# Changing File Pointer Location (ifstream)

- Rather than read sequentially in a file we often need to jump around to particular byte locations
- ifstream.seekg()
  - Go to a particular byte location
  - Pass an offset relative from current position or absolute byte from start or end of file
- ifstream.tellg()
  - Return the current location's byte-offset from the beginning of the file

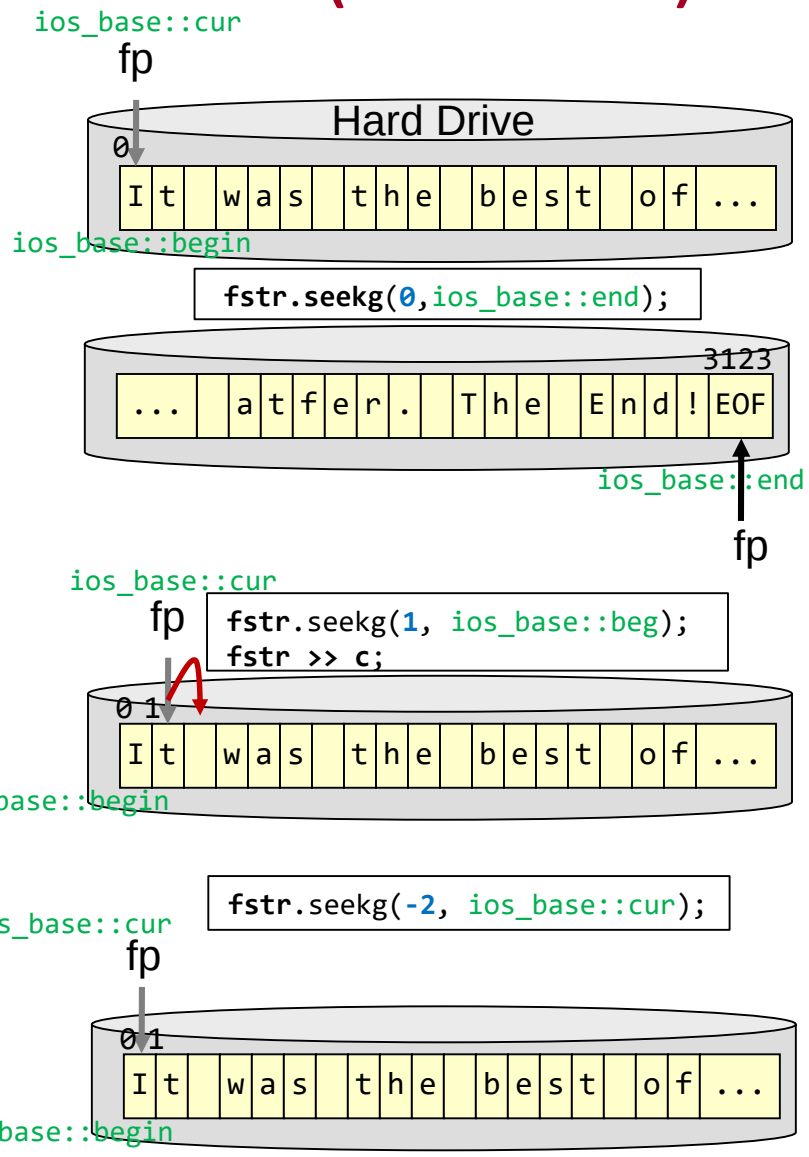
```
int main(int argc, char *argv[])
{
  int size; char c;
  ifstream fstr("stuff.txt");

  fstr.seekg(0,ios_base::end);
  size = fstr.tellg();
  cout << "File size (bytes)=" << size << endl;

  fstr.seekg(1, ios_base::beg);
  cout << "2nd byte in file is ";
  fstr >> c;
  cout << c << endl;
  fstr.seekg(-2, ios_base::cur);
  cout << "1st byte in file is ";
  fstr >> c;
  cout << c << endl;
  fstr.close();
  return 0;
}
```

**2nd arg. to seekg()**

ios_base::beg = pos. from beginning of file
ios_base::cur = pos. relative to current location
ios_base::end = pos. relative from end of file
(i.e. 0 or negative number)

# Changing File Pointer Location (ifstream)

```
int main(int argc, char *argv[])
{
  int size; char c;
  ifstream fstr("stuff.txt");

  fstr.seekg(0,ios_base::end);
  size = fstr.tellg();
  cout << "File size (bytes)=" << size << endl;

  fstr.seekg(1, ios_base::beg);
  cout << "2nd byte in file is ";
  fstr >> c;
  cout << c << endl;
  fstr.seekg(-2, ios_base::cur);
  cout << "1st byte in file is ";
  fstr >> c;
  cout << c << endl;
  fstr.close();
  return 0;
}
```




**2nd arg. to seekg()**

ios_base::beg = pos. from beginning of file
ios_base::cur = pos. relative to current location
ios_base::end = pos. relative from end of file
(i.e. 0 or negative number)

# Changing File Pointer Location (ofstream)

- We can seek and tell in an ofstream
- ofstream.seekp()
  - Go to a particular byte location
  - Pass an offset relative from current position or absolute byte from start or end of file
- ofstream.tellp()
  - Return the current location's byte-offset from the beginning of the file

**2nd arg. to seekg()**

ios_base::beg = pos. from beginning of file
ios_base::cur = pos. relative to current location
ios_base::end = pos. relative from end of file
(i.e. 0 or negative number)

# FILE I/O LAB OVERVIEW

# Lab 7 Overview

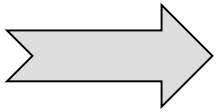
- Modify the word scramble game done in class to allow for a word bank (choice of words to use) to be read in from a file

# Lab 7

- Open up the file and check if it succeeds

wordbank.txt


**Ifstream object**

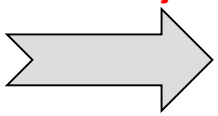


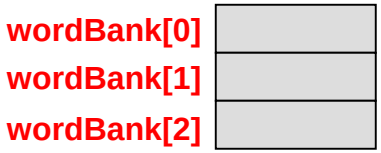
# Lab 7

- Open up the file and check if it succeeds
- Read the number of words expected, check if it succeeds, and if so, allocate the wordBank array of pointers

wordbank.txt


Ifstream object

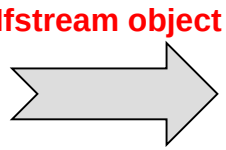


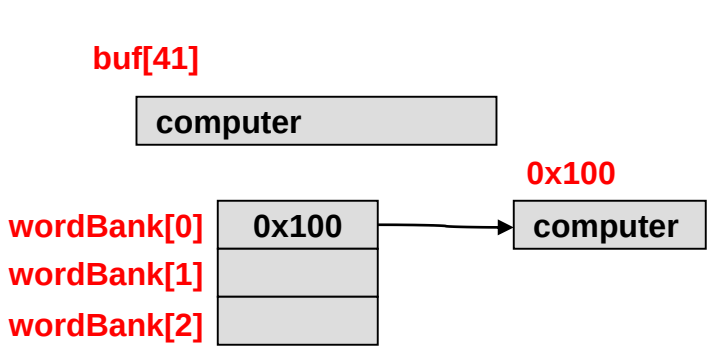


# Lab 7

- In a loop read in each word into a buffer and then allocate some memory to hold that word and copy it to that memory

wordbank.txt

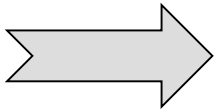



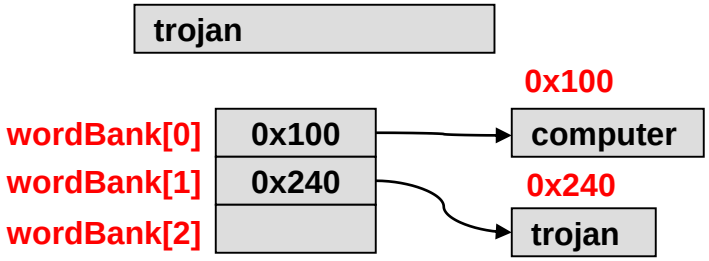
# Lab 7

- In a loop read in each word into a buffer and then allocate some memory to hold that word and copy it to that memory

wordbank.txt


Ifstream object



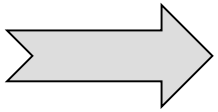


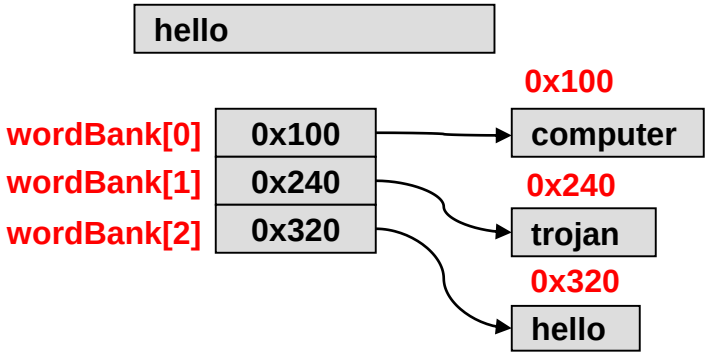
# Lab 7

- In a loop read in each word into a buffer and then allocate some memory to hold that word and copy it to that memory

wordbank.txt


Ifstream object





You are not responsible for this material

# BINARY FILE I/O

# Binary File I/O

- read() – member of ifstream
  - Pass a pointer to where you want the data read from the file to be placed in memory (e.g. `&x` if x is a single int or `data` if data is an array)…this pointer should be cast to a char *
  - Pass # of bytes you want to read = number_of_elements * size_of_element
- write() – member of ofstream
  - Same argument scheme as read()

```
int main(int argc, char *argv[])
{
  int x;
  double data[10];
  ifstream ifile;

  ifile.open("stuff.dat",ios::binary);
  if ( ifile.fail()){
    cerr << "File doesn't exist\n";
    return 1;
  }
  ifile.read(static_cast<char *>(&x), 1*sizeof(int));
  ifile.read(static_cast<char *>(data), 10*sizeof(double));

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

# stdin, stdout, stderr

- Most OS'es map console I/O (keyboard and monitor I/O) to 3 predefined FILE pointers:
  - stdin (input from keyboard) = cin
  - stdout (output to monitor) = cout
    - Normal output
  - stderr (output to monitor) = cerr
    - Exception / error information
  - Unix/Linux can allow you to redirect stdout vs. stderr separately
    - > ./prog > log_of_stdout.txt
    - > ./prog >& log_of_stderr.txt

```
int main(int argc, char *argv[])
{
  char first_char;
  char first_line[80];

  // read char from keyboard
  cin << first_char;

  // read entire line of text from
  //  keyboard
  cin.getline(first_line, 80);

  // echo line back to stdout
  cout << first_line;

  // output to stderr
  cerr << "I had an error." << endl;

  return 0;
}
```

# BACKGROUND ON C FILE I/O (NOT COVERED AFTER FALL 2013)

You are not responsible for this material

# C STYLE I/O

# FILE * variables

- To access files, C (with the help of the OS) has a data type called 'FILE' which tracks all information and is used to access a single file from your program
- You declare a pointer to this FILE type (FILE *)
- You "open" a file for access using fopen()
  - Pass it a filename string (char *) and a string indicating read vs. write, text vs. binary
  - Returns an initialized file pointer or NULL if there was an error opening file
- You "close" a file when finished with fclose()
  - Pass the file pointer
- Both of these functions are defined in stdio.h

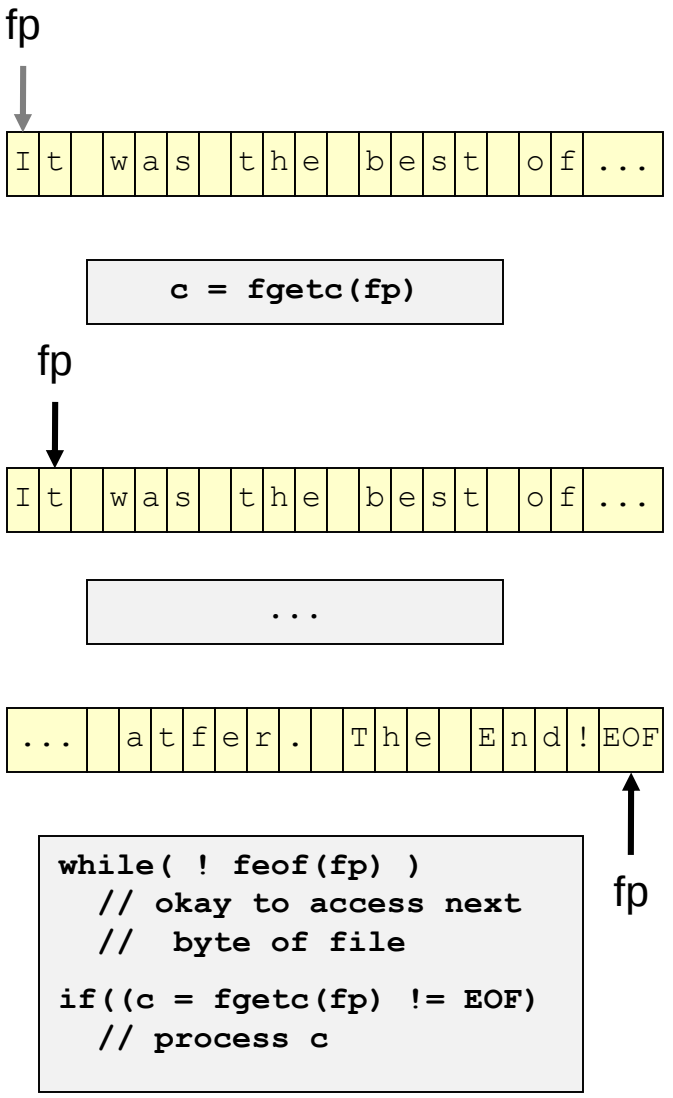
```
int main(int argc, char *argv[])
{
  char first_char;
  char first_line[80];
  FILE *fp;

  fp = fopen("stuff.txt","r");
  if (fp == NULL){
    printf("File doesn't exist\n");
    exit(1)
  }
  // read first char. of file
  first_char = fgetc(fp);
  // read thru first '\n' of file
  fgets(first_line, 80 ,fp);

  fclose(fp);
  return 0;
}
```

**Second arg. to fopen()**

"r" / "rb" = read mode, text/bin file
"w" / "wb" = write mode, text/bin file
"a" / "ab" = append to end of text/bin file
"r+" / "r+b" = read/write text/bin file
others…

# File Access

- Many file I/O functions
  - Text file access:
    - fprintf(), fscanf()
    - fputc(), fgetc(), fputs(), fgets()
  - Binary file access:
    - fread(), fwrite()
- Your file pointer (FILE * var) implicitly keeps track of where you are in the file
- `EOF` constant is returned when you hit the end of the file or you can use feof() which will return true or false.



# Text File Input

- fgetc()
  - Get a single ASCII character
- fgets()
  - Get a line of text or a certain number of characters (up to and including '\n')
  - Stops at EOF…If EOF is first char read then the function returns NULL
  - Will append the NULL char at the end of the characters read
- fscanf()
  - Read ASCII char's and convert to another variable type
  - Returns number of successful items read or 'EOF' if that is the first character read

# Text File Output

- fputc()
  - Write a single ASCII character to the file
- fputs()
  - Write a text string to the file
- fprintf()
  - Write the resulting text string to the file

# Binary File I/O

- fread()
  - Pass a pointer to where you want the data read from the file to be placed in memory (e.g. `&x` if x is an int or `data` if data is an array)
  - Pass the number of 'elements' to read then pass the size of each 'element'
  - # of bytes read = number_of_elements * size_of_element
  - Pass the file pointer
- fwrite
  - Same argument scheme as fread()

```
int main(int argc, char *argv[])
{
  int x;
  double data[10];
  FILE *fp;

  fp = fopen("stuff.txt","r");
  if (fp == NULL){
    printf("File doesn't exist\n");
    exit(1)
  }
  fread(&x, 1, sizeof(int), fp);
  fread(data, 10, sizeof(double),fp);

  fclose(fp);
  return 0;
}
```

# Changing File Pointer Location

- Rather than read/writing sequentially in a file we often need to jump around to particular byte locations
- fseek()
  - Go to a particular byte location
  - Can be specified relative from current position or absolute byte from start or end of file
- ftell()
  - Return the current location's byte-offset from the beginning of the file

```
int main(int argc, char *argv[])
{
  int size;
  FILE *fp;

  fp = fopen("stuff.txt","r");
  if (fp == NULL){
    printf("File doesn't exist\n");
    exit(1)
  }
  fseek(fp,0,SEEK_END);
  size = ftell(fp);

  printf("File is %d bytes\n", size);


  fclose(fp);
  return 0;
}
```

**Third arg. to fseek()**

SEEK_SET = pos. from beginning of file
SEEK_CUR = pos. relative to current location
SEEK_END = pos. relative from end of file
(i.e. negative number)