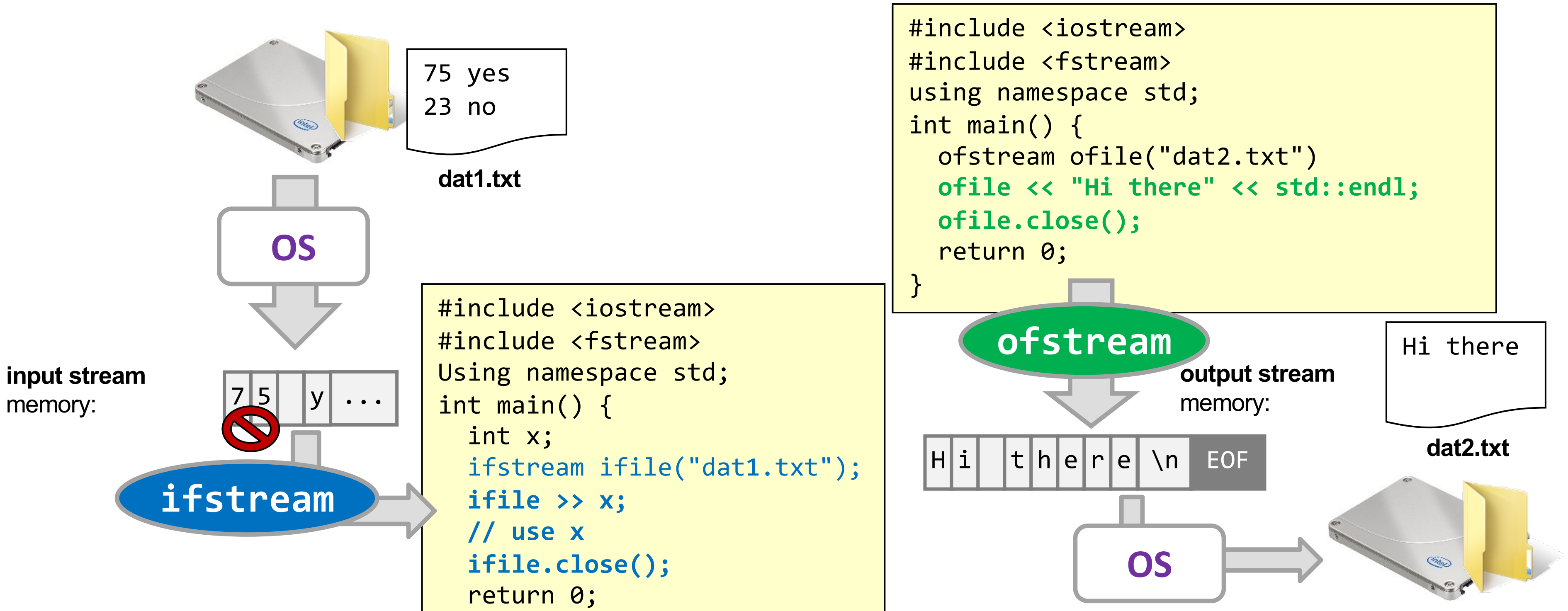
CS 103 Unit 5h – File I/O Part 2

CSCI 103L Teaching Team

File Streams

- C++ leverages the SAME interface that cin and cout provide to (via inheritance):
  - Read data IN from a file (like `cin`, but data comes from a **file** not the keyboard) and
  - Write data OUT to a file (like `cout`, but data goes to a **file** not the terminal).
- The counterpart to `cin` is an `ifstream` object
- The counterpart to `cout` is an `ofstream` object



```
#include <iostream>
#include <fstream>
Using namespace std;
int main() {
  int x;
  ifstream ifile("dat1.txt");
  ifile >> x;
  // use x
  ifile.close();
  return 0;
}
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
  ofstream ofile("dat2.txt")
  ofile << "Hi there" << std::endl;
  ofile.close();
  return 0;
}
```

How your program can directly access data in files

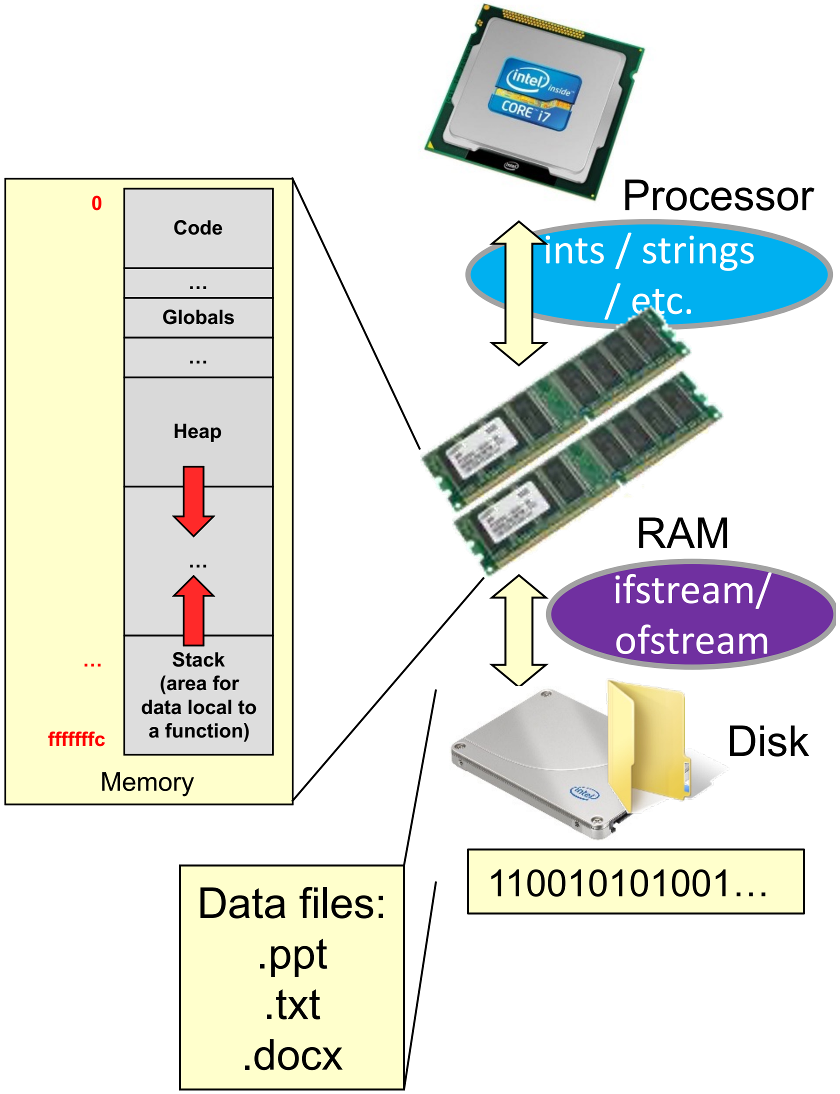
DIRECT FILE I/O USING C++ STREAMS

Important Fact

- For your program to operate on data in a file…
- …you **MUST** read it into a C/C++ variable before processing it
- Everything we will see subsequently is simply how to get data into a variable
  - After that we can just process it normally

Computer Organization

- Why can't we just process data in a file directly?
- Because the processor can only talk directly to RAM / memory
  - It needs “translation” to access data on the hard drive or other disk
- All code and data resides in RAM
  - RAM stores all variables / data that your program accesses
- How do we access files
  - The C++ library and the OS provide routines to perform the translation to read/write data from RAM to a file.



Starting File I/O

- Just like with 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

Two Kinds of Files: Binary and Text

- 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

Text File I/O

- Text file I/O (what we've learned previously) can simply use **ifstream** and **ofstream** objects and `operator>>`, `operator<<`, and `getline()`
  - Can do anything cin/cout can do
- Must include `<fstream>`

```
#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;
 }
```

input.txt

```
5 -3.5
```

output.txt

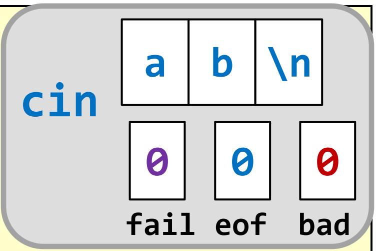
```
Int from file is 5
Double from file is -3.5
```

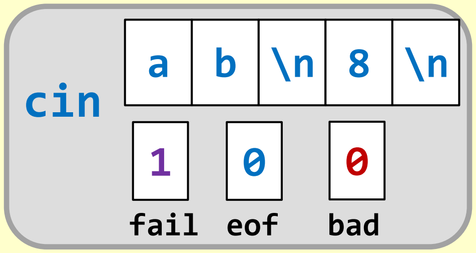
RECOVERING FROM ERRORS

Input Stream Error Checking

- We use the fail() member function of input streams to DETECT ERRORS
- When an operation fails, the input stream sets an internal flag bit (FAIL bit)
  - But this bit STAYS ON even if subsequent operations succeed!
  - We must CLEAR that fail bit using `cin.clear()`
- However, the data in the input stream stays there and will continue to cause us to fail if we don't throw it away using the `cin.ignore()` function
  - Takes a maximum number of characters to throw away or a delimeter to stop on ('\n')
  - E.g. `cin.ignore(256, '\n')`

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

int main ()
{
  int x;
  cout << "Enter an int: " << endl;

  cin >> x;  // What if the user enters:
             //    "ab"


  // 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 << "Nice!  X = " << x << endl;
  return 0;
 }
```

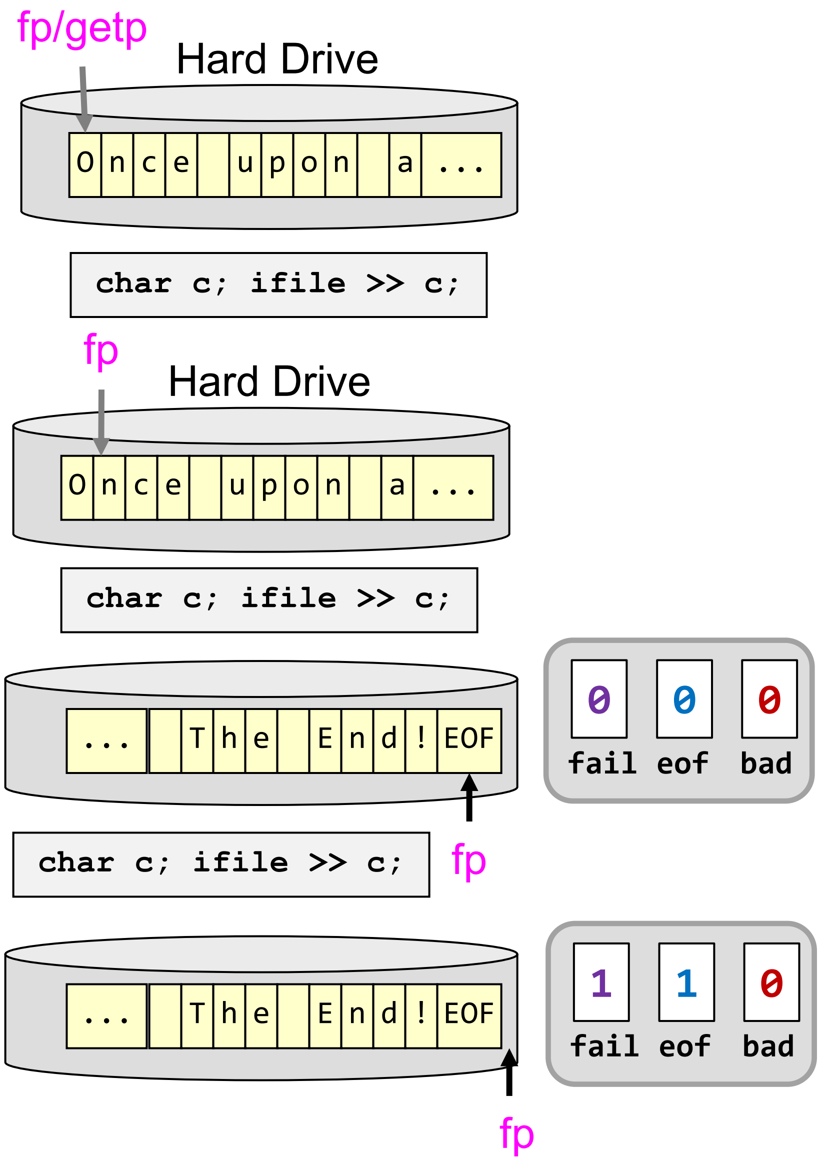




FILE LOCATION/POINTERS & INPUT OPERATORS

File Streams and EOF

- Your ifstream object implicitly keeps track of where you are in the file using a file pointer (fp) or get pointer (getp)
- 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
- Input streams also allow you to check if you've read the EOF character by calling an `eof()` function, but fail will be set when eof is and so it's easier to just use `fail()`



```
char c; ifile >> c;
```

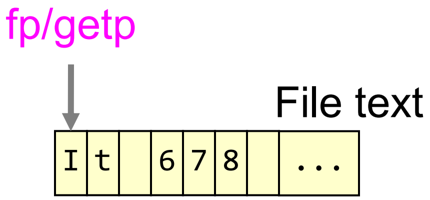
```
char c; ifile >> c;
```

```
char c; ifile >> c;
```

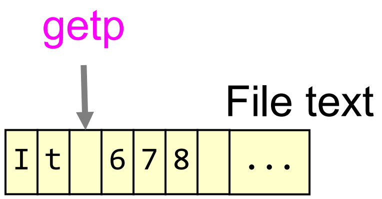
operator>>

- 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 operator>>
- To get raw data from the file (including whitespaces) use the get() function

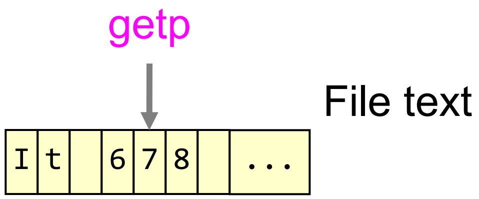
```
ifstream ifile("data.txt");
```



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



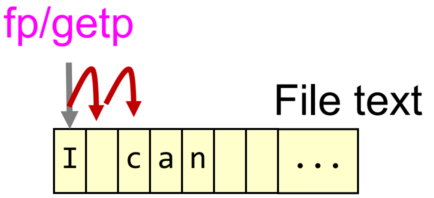
```
char x; ifile >> x;
// skips space & gives x='6'
```



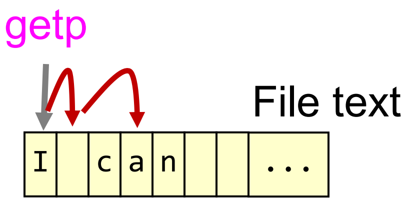
operator>> vs. get vs. peek

- 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/getp" pointer on to the next character, use `ifstream::peek()` function
  - Returns the character at the "fp/getp" but does NOT move "fp/getp" on

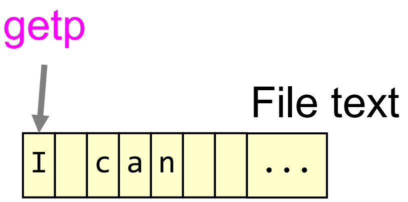
```
ifstream ifile("data.txt");
```



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



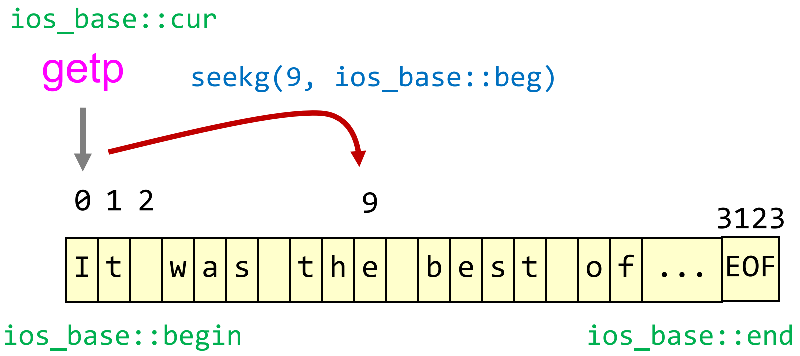
```
ifile >> c; // returns 'I'
ifile >> c; // skips space and
            // returns 'c'
```



```
c = ifile.peek(); // returns 'I'
   // and doesn't move to next char
```

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
  - To specify what the offset is relative to, use one of ios_base::beg/cur/end
- `ifstream::tellg()`
  - Return the current location's byte-offset from the beginning of the file



ios_base::cur

getp

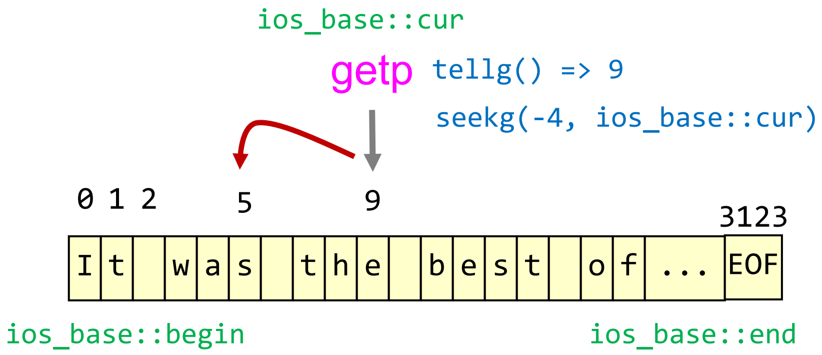
seekg(9, ios_base::beg)

0 1 2 9 3123

I	t		w	a	s		t	h	e		b	e	s	t		o	f	...	EOF

ios_base::begin

ios_base::end



ios_base::cur

getp

tellg() => 9

seekg(-4, ios_base::cur)

0 1 2 5 9 3123

I	t		w	a	s		t	h	e		b	e	s	t		o	f	...	EOF

ios_base::begin

ios_base::end

2nd arg. to seekg()

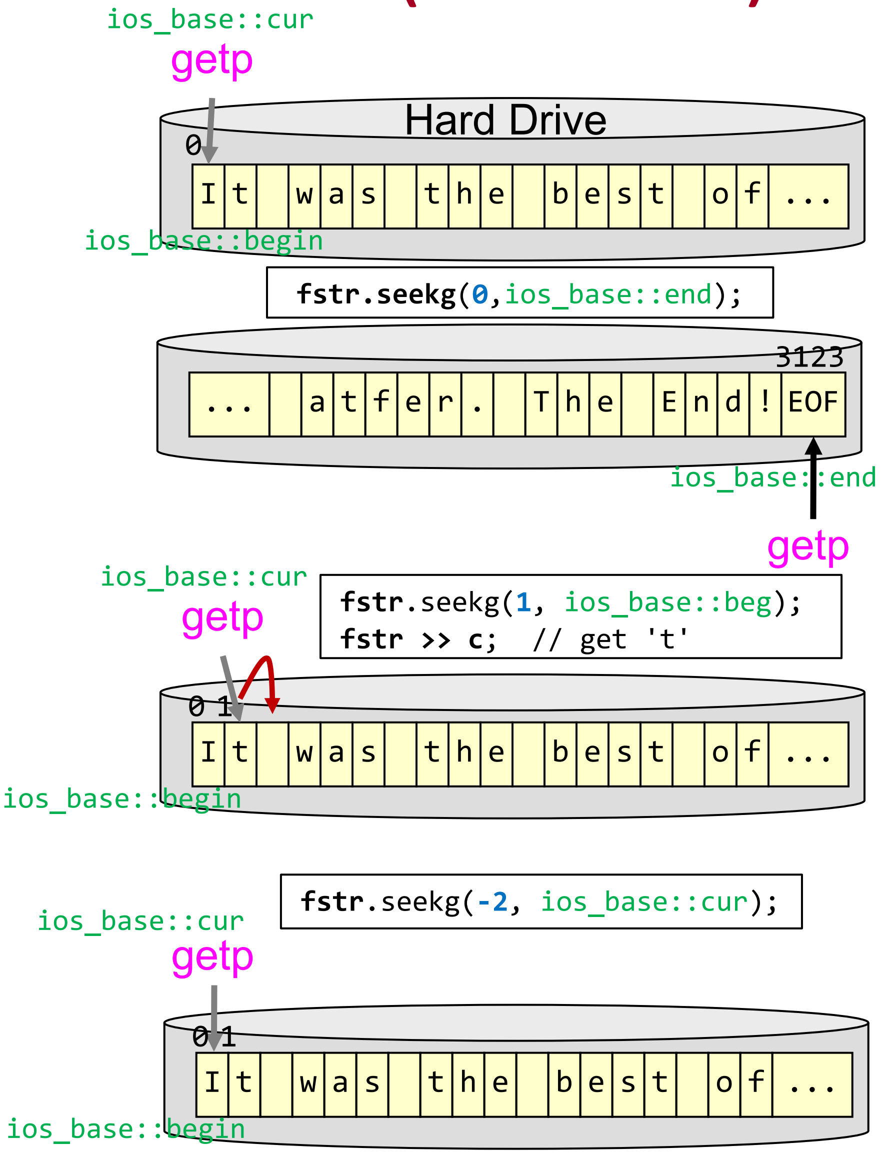
ios_base::beg = pos. from beginning of file
ios_base::cur = pos. relative to current location
ios_base::end = pos. relative from end of file (i.e. 0 or negative number)

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)



```
fstr.seekg(0,ios_base::end);
```

```
fstr.seekg(1, ios_base::beg);
fstr >> c;  // get 't'
```

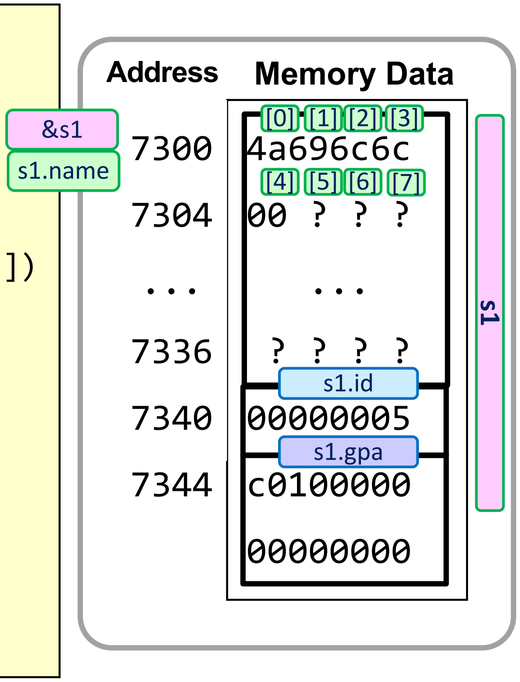
```
fstr.seekg(-2, ios_base::cur);
```

BINARY FILE I/O

Binary vs. Text File I/O

- Binary file content takes the literal bits from memory/RAM and saves it to a file
- Text file content is the ASCII representation of the data (as it would be printed).

```
struct Student {
   char name[40];
   int major;
   double gpa;
};

int main(int argc, char *argv[])
{
  Student s1;
  strcpy(s1.name, "Jill");
  s1.major = 5;
  s1.gpa = 3.7;

  return 0;
}
```



Address	Memory Data
7300	4a696c6c
7304	00 ? ? ?
...	...
7336	? ? ? ?
7340	00000005
7344	c0100000
	00000000

Binary File Content

```
4a696c6c
00 ? ? ?
   ...
 ? ? ? ?
00000005
c0100000
00000000
```

```
Jill 5 4.0
```

Text File Content

```
4A 69 6C 6C 20 35
20 34 2E 30 0A
```

Binary File I/O Functions

- When opening the file, include the "binary" mode flag
  - `open(const char* filename, ios_base::openmode mode)`
  - In addition, to the filename argument, provide `ios::binary`
- To **write** data to a binary file use the `write()` function
  - `ostream& ostream::write (const char* s, streamsize n);`
  - s = pointer to the starting address of the data to write to the file
  - should be cast to a char*
  - n = the number of bytes to be written = number_of_elements * size_of_element
- To **read** data from a binary file use the `read()` function
  - `istream& istream::read (char* s, streamsize n);`
  - s = a pointer to where you want the data read from the file to be placed in memory…this pointer should be cast to a char*
  - n = Number of bytes you want to read

Copy a File

```
#include <iostream>
#include <fstream>      // std::ifstream, std::ofstream
using namespace std;

int main () {
  std::ifstream infile ("src.txt",ios::binary);
  std::ofstream outfile ("copy.txt",ios::binary);

  infile.seekg (0,infile.end); // get size of file
  long size = infile.tellg();
  infile.seekg (0);

  // allocate memory for file content
  char* buffer = new char[size];

  // read content of infile
  infile.read (buffer,size);

  // write to outfile
  outfile.write (buffer,size);

  // release dynamically-allocated memory
  delete[] buffer;

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

https://cplusplus.com/reference/ostream/ostream/write/

Binary File I/O

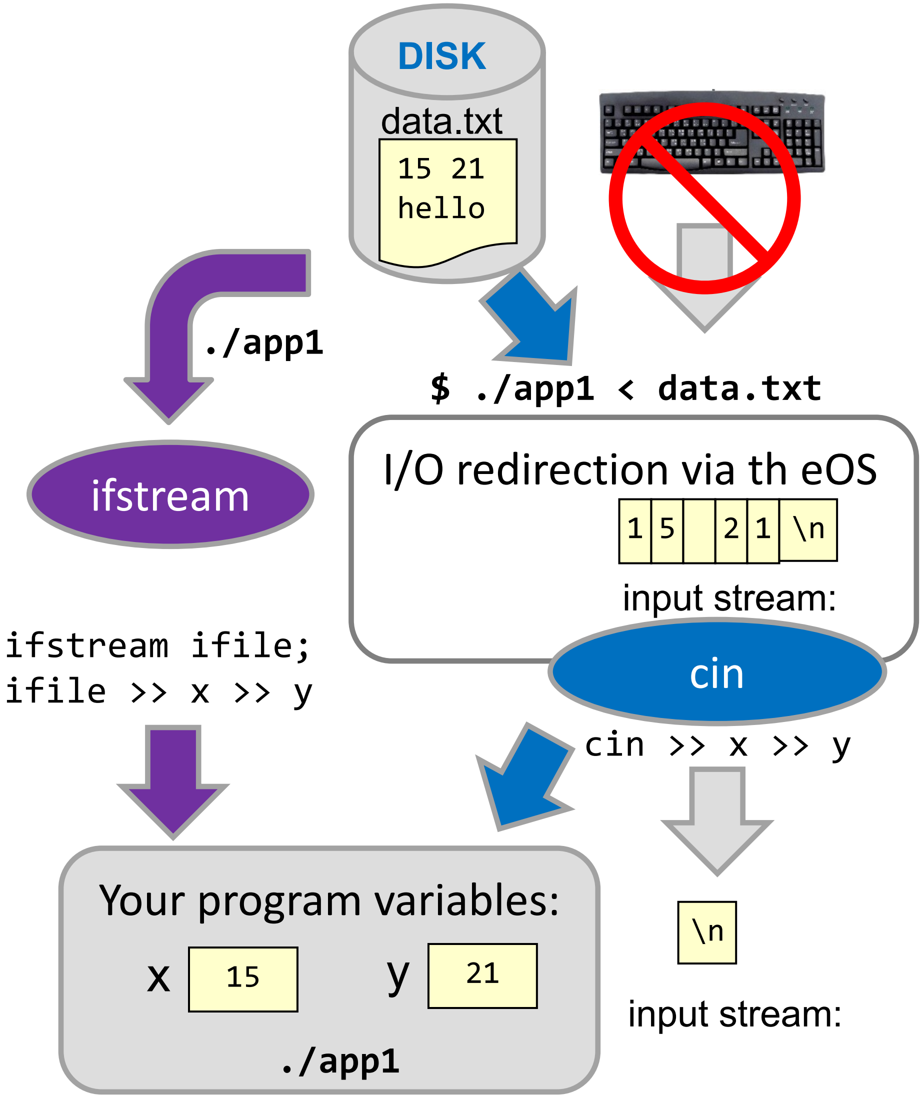
- write() – member of ofstream
  - Pass a pointer to the starting location of the data to write to the file (should be cast to a char*) and the number of bytes to be written = number_of_elements * size_of_element
- read() – member of ifstream
  - Pass a pointer to where you want the data read from the file to be placed in memory…this pointer should be cast to a char*
  - Pass # of bytes you want to read

```
struct Student {
   char name[40];
   int major;
   double gpa;
};

void saveToFile(const char* fname, Student* data) {
  ofstream ofile(fname, ios::binary);
  ofile.write(static_cast<char *>(data), 100*sizeof(Student));
  ofile.close();
}
void readFromFile(const char* fname, Student* data) {
  ifstream ifile(fname, ios::binary);
  ifile.read(static_cast<char *>(data), 100*sizeof(Student));
  ifile.close();
}
int main(int argc, char *argv[])
{
  Student stu[100];
  // initialize and fill the 100 Student objects
  saveToFile("class.dat", stu);

  Student duplicate[100];
  readFromFile("class.dat", duplicate);
  return 0;
}
```

I/O REDIRECTION

Overview

- Two methods for file I/O
  - **File streams** (`ifstream` and `ofstream`) are part of the C++ library and perform file I/O directly through a cin- and cout-like interface **(Covered in Unit 3)**
  - **I/O redirection**: The OS reads or writes data to/from a file by controlling cin & cout
    - The program just performs normal cin and cout commands
    - **Covered in this unit**



File I/O Options

- A second method (other than `ifstream` and `ofstream` objects) is to use an OS mechanism called **I/O Redirection**
  - All general operating systems support this mechanism.
- The OS can:
  - Redirects the contents of a file into **stdin** (i.e. **cin**)
  - Redirect the output sent to **stdout** (i.e. **cout**) to a file rather than the terminal

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
- | pipe output of first program to second
  - **stdout** of first program is then used as **stdin** of next program

Redirection & Pipe Examples

- `$ ./shapes < input.txt`
  - Redirects contents of input.txt to **stdin** (i.e. cin) in HW2 shapes program
- Codio Demo
  - Go to Codio and find the Exercise 6 – I/O Redirection
- From the terminal, compile the programs
  - `$ 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

BACKGROUND ON C FILE I/O (NOT COVERED)

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

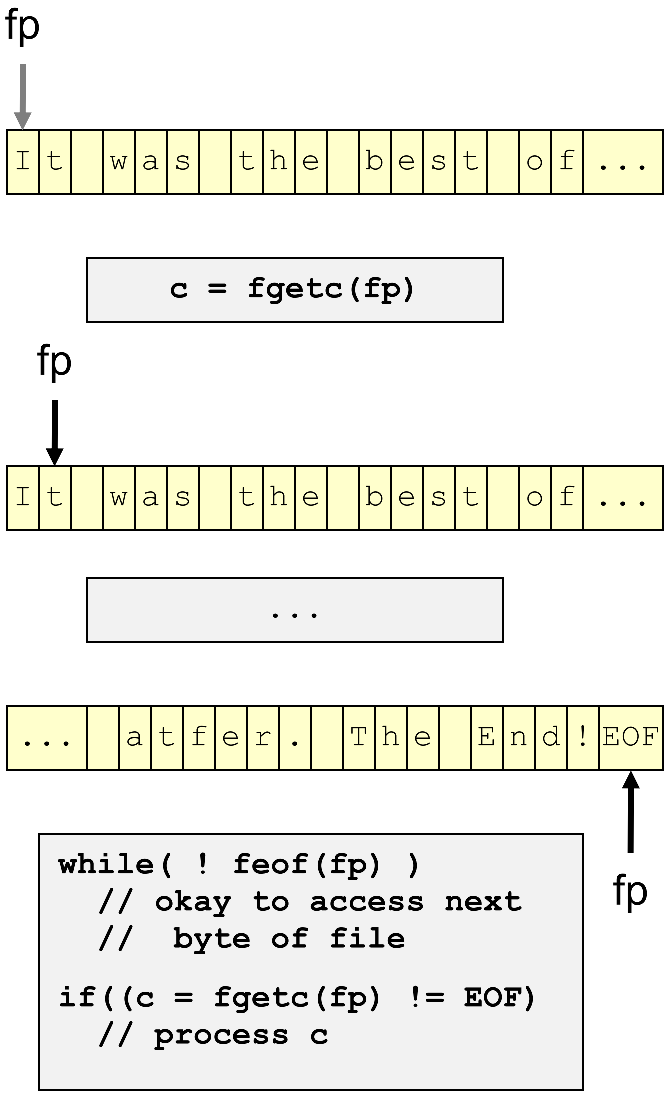
```
int main(int argc, char *argv[])
{
  char first_char;
  char first_line[80];
  FILE *fp;

  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.



fp

I	t		w	a	s		t	h	e		b	e	s	t		o	f	...

```
c = fgetc(fp)
```

fp

I	t		w	a	s		t	h	e		b	e	s	t		o	f	...

```
...
```

...		a	t	f	e	r	.		T	h	e		E	n	d	!	EOF

fp

```
while( ! feof(fp) )
  // okay to access next
  //  byte of file

if((c = fgetc(fp) != EOF)
  // process c
```

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)