File Input and Output

PHYS2G03

© James Wadsley,

McMaster University

File I/O

- Modern science produces huge amounts of data
- e.g. Experiments: Genomics, Supercolliders, Astronomy, Social Science, Health …
- e.g. Simulations: Climate, Astrophysics, Fluid Dynamics, Physics, Molecular Dynamics …

- Data is stored in files

Data Formats

Data in files can be

- 1) Human readable: plain text

  e.g. what std::cout makes

- 2) Binary/Raw data

  e.g. Same format as computer memory

Text Files

```
#include <iostream>
```

provides cout and cin for terminals

Programs can generate plain text using std::cout and read from the keyboard using std::cin

```
cp –r /home/2G03/fileio  ~/
cd fileio
gedit cout.cpp
```

cout.cpp: print 10 random numbers

```
#include <iostream>
#include <stdlib.h>

int main() {

  int i;
  const int n=10;

  std::cout << n << "\n"; // Write size of data

  for (i=0;i<n;i++) { // Write data (random numbers)
    std::cout << float(rand())/RAND_MAX << "\n";
  }
}
```

cin.cpp: read in n numbers

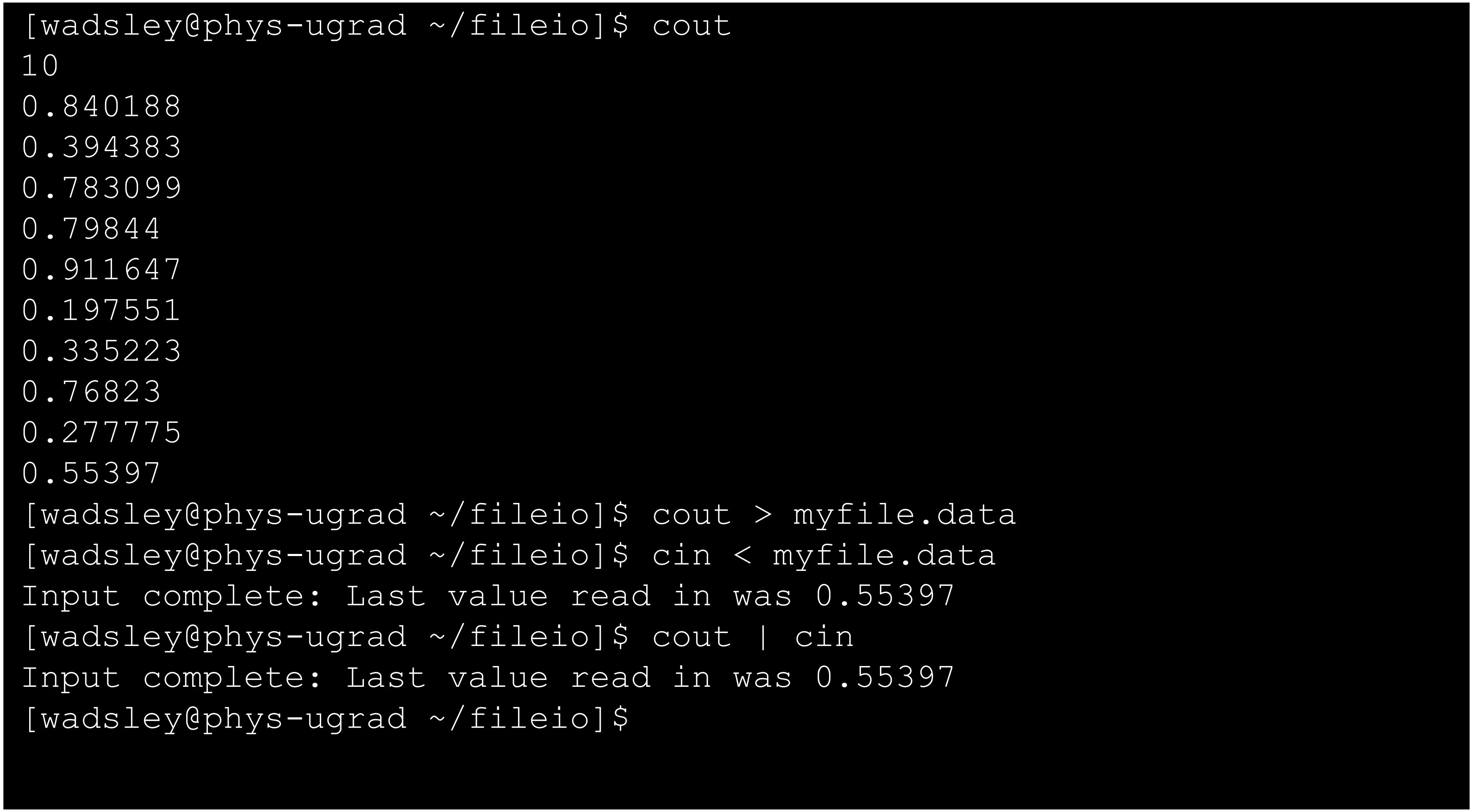
```
#include <iostream>

int main() {

  int i;
  int n;

  // note -- no white space like "\n" or " " for reading!
  std::cin >> n;   // read size

  float inputdata[n]; // Dynamically allocate array

  for (i=0;i<n;i++) { // Read data
    std::cin >> inputdata[i];
  }

  std::cout << "Input complete: Last value read in was " << inputdata[n-1] << "\n";
}
```

cin and cout

Try:

```
make cout
cout
cout > file.data          Put output into a file
more file.data
make cin
cin < file.data           Read input from a file
cout | cin                Pipe output from cout to cin
```

cout, cin > < and pipe |



```
[wadsley@phys-ugrad ~/fileio]$ cout
10
0.840188
0.394383
0.783099
0.79844
0.911647
0.197551
0.335223
0.76823
0.277775
0.55397
[wadsley@phys-ugrad ~/fileio]$ cout > myfile.data
[wadsley@phys-ugrad ~/fileio]$ cin < myfile.data
Input complete: Last value read in was 0.55397
[wadsley@phys-ugrad ~/fileio]$ cout | cin
Input complete: Last value read in was 0.55397
[wadsley@phys-ugrad ~/fileio]$
```

Working with files

cin and cout are not very flexible. In general you want to be able to open and close files with a program to read or write from them – without needing a terminal

```
#include <fstream>
```

Extends io to use files instead of terminal as source of data

fstream

Instead of using cout and cin, you make a new object that represents the file

#include <iostream>   Must include iostream too!
#include <fstream>

ofstream outputfile;   a stream object for output
ifstream inputfile;   a stream object for input

fout.cpp: write to “output.data”

```
#include <iostream>
#include <fstream>

#include <stdlib.h>
int main() {
  int i;
  const int n=10;

  std::ofstream outputfile;

  outputfile.open( "output.data" );
  outputfile << n << "\n"; // Write size of data

  for (i=0;i<n;i++) { // Write data (random numbers)
    outputfile << float(rand())/RAND_MAX << "\n";
  }

  outputfile.close();
}
```

I want to write to the file – so I use ofstream

ofstream

Direct replacement for std::cout

Extra step: open a file first!

Try:

make fout

fout

more output.data

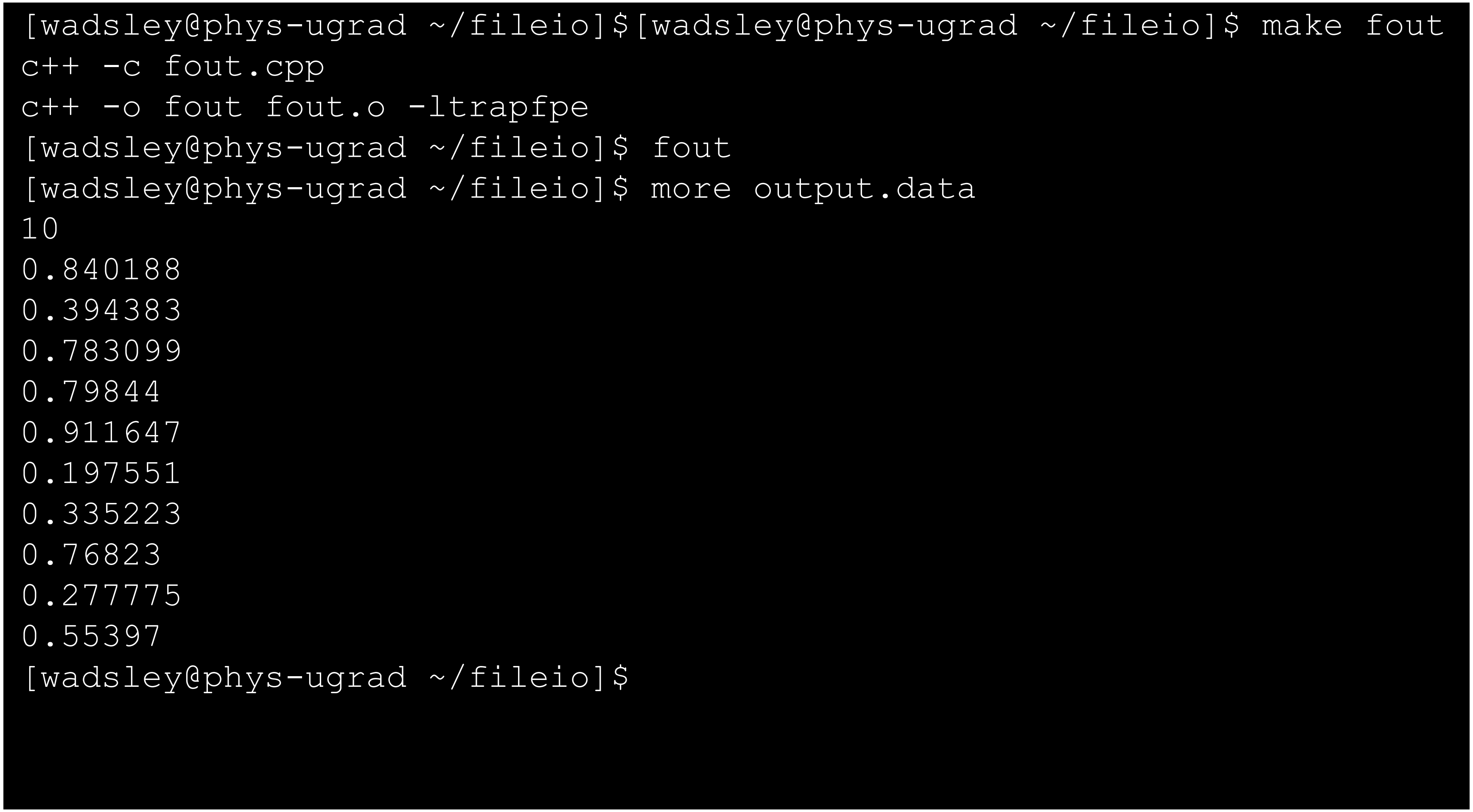
cin < output.data

fout writes directly to a file

The program chooses the filename

Unlike cout, I can open as many files at once as I like

fout



```
[wadsley@phys-ugrad ~/fileio]$[wadsley@phys-ugrad ~/fileio]$ make fout
c++ -c fout.cpp
c++ -o fout fout.o -ltrapfpe
[wadsley@phys-ugrad ~/fileio]$ fout
[wadsley@phys-ugrad ~/fileio]$ more output.data
10
0.840188
0.394383
0.783099
0.79844
0.911647
0.197551
0.335223
0.76823
0.277775
0.55397
[wadsley@phys-ugrad ~/fileio]$
```

fin.cpp: read from “output.data”

```
#include <iostream>
#include <fstream>

int main() {
  int i;
  int n;
  std::ifstream inputfile;
  inputfile.open("output.data" );
  // note -- no white space like "\n" or " " for reading!
  inputfile >> n;   // read size
  float inputdata[n]; // Dynamically allocate array

  for (i=0;i<n;i++) { // Read data
    inputfile >> inputdata[i];
  }
  inputfile.close();
  std::cout << "Input complete: Last value read in was " << inputdata[n-1] << "\n";
}
```

ifstream

Direct replacement for std::cin

Extra step: open a file first (file must exist!)

Try:

make fin

fout

fin   Note: fin is hardwired to read output.data

fin_ask.cpp   Ask the user

```
#include <iostream>
#include <fstream>
#include <string>
int main() {
  int i, n;
  std::string filename;
  std::cout << "What file do you want to open?\n";
  std::cin >> filename;
  std::ifstream inputfile;
  inputfile.open(filename.c_str());
```

Note: open() needs a char string: convert filename using .c_str() function

```
  inputfile >> n;   // read size
  float inputdata[n]; // Dynamically allocate array
  for (i=0;i<n;i++) { // Read data
    inputfile >> inputdata[i];
  }
  inputfile.close();
  std::cout << "Input complete: Last value read in was " << inputdata[n-1] << "\n";
```

fstream: Did it work?

fstream objects have extra functions to test if things worked.

e.g.

```
#include <iostream>
#include <fstream>

int main () {
  std::ofstream myfile ("example.txt");
  if (myfile.is_open()) {
      // Test that it opened
  }
  else std:: cout << "Unable to open file”;
```

fin_ask.cpp Test if open worked

```
#include <iostream>
#include <fstream>
#include <string>
int main() {
  int i,n;;
  std::string filename;
  std::cout << "What file do you want to open?\n";
  std::cin >> filename;

  std::ifstream inputfile;
  inputfile.open(filename.c_str());
  if (!inputfile.is_open()) {
    std::cout << "Could not open file: " << filename << "\n";
    return 1;
  }
```

ifstream

fin_ask.cpp uses cin to ask the user for a filename

Try:

make fin_ask

fin_ask ask it to open output.data

fstream basic functions

open("name")	open a file
close()	close the file
is_open()	Is it open?
bad()	Something went wrong?
fail()	Failed to read/write or bad?
eof()	Are we at the end of the file?
good()	OK to keep reading/writing?

Example of a problem

Try:

fin_ask Tell it: short.data

short.data sets size 10 but only has 4 numbers to read

make fin_test

fin_test Tell it: short.data

This program checks after each read

fin_test.cpp Test if read worked

```
inputfile >> n;   // read size
float inputdata[n]; // Dynamically allocate array

for (i=0;i<n;i++) { // Read data
  inputfile >> inputdata[i];
  if (inputfile.fail()) {
    std::cout << "Read incomplete: at line: " << i << "\n";
    if (i>0) std::cout <<"Last good data value " <<
                              inputdata[i-1] << "\n";
    return 1;
  }
}
inputfile.close();
```

File I/O

You can open as many files are you want at once.

You can control the way the files are used and how the system responds if files do or don’t exist, can’t be written etc…

Options for fstream

Open can be used with 2 arguments

e.g.

```
ifstream  myfile;
myfile.open(“somefile.dat”,  mode );
```

With mode you can specify what do do about the file if it exists, expectations about data format and so on

Mode options

- std::ios::in Open for input operations.
- std::ios::out Open for output operations.
- std::ios::binary Open in binary mode.
- std::ios::ate Set the initial position at the end of the file.

- std::ios::app All output operations are appended to end of file (only for output)
- std::ios::trunc If the file opened for output operations, delete previous

Using mode options

```
std::ofstream  myfile;
myfile.open(“somefile.dat”,  std::ios::app );
```

Open file to append to end of file (don’t overwrite what’s already there)

Default is to overwrite from the start

fstream

fstream is the general version of a file stream

- You can create a stream for input or output as you wish

```
fstream  myfile;
myfile.open(“somefile.dat”, std::ios::in );
```

- ifstream assumes input (reading a file)
- ofstream assumes output (writing a file)

Binary output

Writing text is only useful if a human wants to read it. For the computer, it requires translating what is in its memory into characters

You can write memory contents directly. It is faster and more efficient (smaller files).

It is also exact – you save precisely what the computer stored in memory

binaryout.cpp

```cpp
#include <iostream>
#include <fstream>
#include <stdlib.h>

int main() {
  int i;
  const int n=10;

  std::ofstream outputfile;
  outputfile.open( "output.bin", std::ios::binary );
  outputfile.write( (char *) &n, sizeof(n) );

  for (i=0;i<n;i++) { // Write data (random numbers)
    float data = float(rand())/RAND_MAX;
    outputfile.write( (char *) &data, sizeof(data) );
  }
  outputfile.close();
}
```

Working with binary

Don’t use << >> operators. They are for translating variables into text

Directly read and write memory from/to a file:

```
  outputfile.open( "output.bin", std::ios::binary );
  outputfile.write( (char *) stuff,  nBytes );
```

C/C++ char variables are equivalent to bytes

The fstream read/write just want a memory location and how many bytes to read/write!

Working with binary

Direct write memory to a file:

Write one integer n:

```
   int n = 42;
  outputfile.write( (char *) &n,  sizeof(n)  );
```

(char *) says “treat the address of n as a pointer to raw bytes”. It doesn’t change the address.

sizeof(n) is how many bytes in an int   (4)

This writes 4 bytes directly from memory to the file

Working with binary

Direct write memory to a file:

Write an array of 100 floats:

```
float a[100];
outputfile.write( (char *) a,  100*sizeof(a[0])  );
```

(char *) => treat array as a pointer to raw bytes

sizeof(a[0]) is how many bytes in an float (4)

Writes 400 bytes directly from memory to the file

```
outputfile.write( (char *) a,  100*sizeof(float)  );
```

also ok

Reading Binary

Direct read memory to a file:

Write an array of 100 floats:

```
float a[100];
outputfile.read( (char *) a,  100*sizeof(a[0])  );
```

(char *) converts array a into a pointer to raw bytes

sizeof(a[0]) is how many bytes in an float (4)

Writes 400 bytes directly from the file to memory

Reading Binary

Direct read memory to a file:

Write an array of n floats:

```
float a[n];
outputfile.read( (char *) a,  n*sizeof(a[0])  );
```

C/C++ lets you decide the array size a the last minute and then read data into that memory

(char *) => treat array a into a pointer to raw bytes

sizeof(a[0]) is how many bytes in an float (4)

Writes 4 n bytes directly from the file to memory

binaryin.cpp

```
#include <iostream>
#include <fstream>

int main() {
  int i;
  int n;

  std::ifstream inputfile;

  inputfile.open("output.bin" );
  inputfile.read( (char *) &n, sizeof(n)); // read size

  float inputdata[n]; // Dynamically allocate array
  inputfile.read( (char *) inputdata, n*sizeof(inputdata[0]));
  inputfile.close();

  std::cout << "Input complete: Last value read in was " << inputdata[n-1] << "\n";
}
```

binaryin.cpp

```
#include <iostream>
#include <fstream>

int main() {
  int i;
  int n;

  std::ifstream inputfile;

  inputfile.open("output.bin" );
  inputfile.read( (char *) &n, sizeof(n)); // read size

  float inputdata[n]; // Dynamically allocate array
  inputfile.read( (char *) inputdata, n*sizeof(inputdata[0]));
  inputfile.close();

  std::cout << "Input complete: Last value read in was " << inputdata[n-1] << "\n";
}
```

Note how compact and efficient this binary read is --- just read all 4n bytes into memory in one go!

Look at the binary files!

Try:

```
make binaryout
binaryout
fout
ls –l output*     Which is bigger?
make binaryin
binaryin
make binaryin_test     testing version
```

Raw text is bigger

```
[wadsley@phys-ugrad fileio]$ ls -l out* short*
-rw-rw-r-- 1 wadsley wadsley 44 Oct  5 02:13 output.bin
-rw-rw-r-- 1 wadsley wadsley 90 Oct  5 02:13 output.data
-rw-rw-r-- 1 wadsley wadsley 20 Oct  5 02:13 short.bin
-rw-rw-r-- 1 wadsley wadsley 38 Oct  5 02:13 short.data
```

If you write more decimal places text (output.data) would be even bigger

Binary data contains exactly what is in memory and no more (output.bin)

Looking at binary: od command

Integers

```
[wadsley@phys-ugrad ~/fileio]$ od -i output.bin
0000000         10  1062672011  1053420687  1061714225
0000020  1061971601  1063870905  1045056232  1051435601
0000040  1061464754  1049507965  1057870074
0000054
```

floats

```
[wadsley@phys-ugrad ~/fileio]$ od -f output.bin
0000000   1.401298e-44   8.401877e-01   3.943829e-01   7.830992e-01
0000020   7.984400e-01   9.116474e-01   1.975514e-01   3.352228e-01
0000040   7.682296e-01   2.777747e-01   5.539700e-01
0000054
```

Compare to text file

```
[wadsley@phys-ugrad ~/fileio]$ more  output.data
10
0.840188
0.394383
0.783099
0.79844
0.911647
```

binaryin_test.cpp

```
inputfile.read( (char *) &n, sizeof(n)); // read size
std::cout << n << " data values to read\n";

float inputdata[n]; // Dynamically allocate array

if (!inputfile.read( (char *) inputdata, n*sizeof(inputdata[0]))) {
  std::cout << "Read incomplete: only read " << inputfile.gcount() <<
                         " bytes \n";
  int nread = inputfile.gcount()/sizeof(inputdata[0]);
  if (nread > 0)
    std::cout <<"Last good data value " << inputdata[nread-1] << "\n";
  return 1;
}
inputfile.close();
```

Testing for success

read() and write() return 0 if there was a problem

```
if (myfile.read( stuff, nBytes )) { // success! }
```

You can also ask how much was read

```
std::cout << myfile.gcount()) << “ bytes read”;
```

This can be converted into a number of values with sizeof

Try binaryin_test   with file: short.bin

binaryin_test.cpp

```
inputfile.read( (char *) &n, sizeof(n)); // read size
std::cout << n << " data values to read\n";

float inputdata[n]; // Dynamically allocate array

if (!inputfile.read( (char *) inputdata, n*sizeof(inputdata[0]))) {
  std::cout << "Read incomplete: only read " << inputfile.gcount() <<
                            " bytes \n";
  int nread = inputfile.gcount()/sizeof(inputdata[0]);
  if (nread > 0)
    std::cout <<"Last good data value " << inputdata[nread-1] << "\n";
  return 1;
}
inputfile.close();
```

Binary:
Big Endian and Little Endian

With text there are conventions on how to order numbers: 100 means one hundred (rather than writing 001)

Files consist of bytes

With raw binary of a 4 byte number do you put the big bytes first or the little ones?

Unfortunately chip makers didn’t agree

endian.cpp

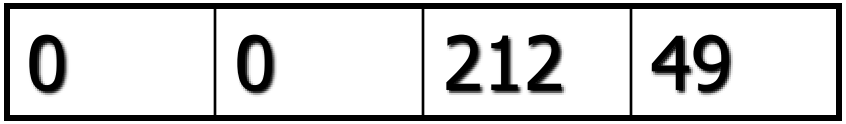
Writes a 4-byte integer to the file: endian.bin
value: 54321

```
#include <iostream>
#include <fstream>
int main() {

  int i=54321;

  std::ofstream outputfile;
  outputfile.open( "endian.bin", std::ios::binary );
  outputfile.write( (char *) &i, sizeof(i) );
  outputfile.close();
}
```

Unformatted: Big Endian and Little Endian

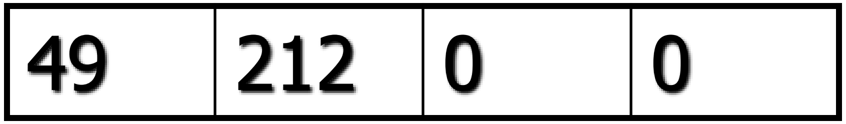
54321=256*212+49

Big Endian



0	0	212	49

Little Endian



49	212	0	0

Many compilers have flags to choose how files are written

Try od –t u1 endian.bin

What endianness is Intel (phys-ugrad)?

Endianness

- Big Endian

Apple*, SUN (Sparc), Internet Standard

- Little Endian

HP (alpha), **Intel, AMD** *macbooks use intel

Internet packets are converted to big-endian regardless of your computer for compatibility

(using the xdr library)

For most purposes you don’t have to care ☺