How to read a Whole Line (forgotten from last weeks Lecture)

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
   cout << "Please enter a line:\n";
   string s;
   getline(cin,s);
   cout << "You entered " << s << ’\n’;
}
```

Working with Streams

- Like its predecessor C, C++ does not directly support input / output (I/O) operations
- However virtually all version of C++ now supply the *standard iostreams library.*
- This library has recently be replaced with the standard iostreams template classes in the Standard C++ Library.
- Not all version of C++ support this as yet.
- File I/O in C++ uses the “File Stream” abstraction similar to that used with cin and cout
- To read from a file we can use the > > extraction operator
- To write to a file we use the < < insertion operator

Standard iostreams Library headers

Header	Purpose
<istream>	Defines the template class basic_istream along with several extractors
<fstream>	Defines several template classes that support iostreams operations on sequences stored in external files
<iomanip>	Defines several manipulators that take a single argument
<ios>	Defines several types and functions basic to the operation of iostreams, including many format manipulators
<iosfwd>	Forward declarations of the iostream classes
<iostream>	Declares global stream objects (such as cin and cout) that control the reading from and writing to the standard stream
<ostream>	Defines the template class basic_ostream along with several insertors and manipulators

Header	Purpose
<istream>	Defines the template class basic_istream along with several extractors
<sstream>	Defines several template classes that support iostream operations on sequences of stored in character arrays - easily converted to and from objects of template class basic_string
<streambuf>	Defines the template class basic_streambuf
<strstream>	Defines several classes that support iostreams operations on sequences of stored in C-style character arrays

Manipulators

- Manipulators are used to manipulate stream objects data in various ways

- They are typically used to format the output of different data types into a user specified way.

- The following table shows the different manipulator types for streams

Flag	Description
boolalpha	Flags a stream to insert or extract Boolean objects as names such as *true* and *false* instead of as numeric values (such as 1 and 0)
dec	Flags a stream to insert or extract integer values in floating- point format (with decimals)
fixed	Flags a stream to insert floating-point values in fixed-point format, without using exponents.
hex	Flags a stream to insert or extract integer values in hexadecimal format.
internal	Flags a stream to pad to a field width as needed by inserting internal fill characters.

Flag	Description
left	Flags a stream to left-justify characters by padding with fill characters on the right
oct	Flags a stream to insert or extract integer values in octal format.
right	Flags a stream to right-justify characters by padding with fill characters to the left
scientific	Flags a stream to insert floating point values in scientific format, using exponents
showbase	Flags a stream to insert a prefix that reveals the base of a generated integer field (for example hex 9 becomes 0x9)
showpoint	Flags a stream to force a decimal point in a floating point field, even when there is no fractional part present
showpos	Flags a stream to insert a plus sign ion a non-negative - generated numeric field.
skipws	Flags a stream to skip white-space before certain extractions
unitbuf	Flags a stream to flush output after each insertion (otherwise an endl must be used)
uppercase	Flags a stream to insert uppercase equivalents of lowercase letters in certain insertions.

Using Manipulators to set flags

- To set the cout manipulator flags we use the following cout methods

```
cout.setf( flag to set); // set a flag
cout.unsetf(flag to un-set); // unset a flag
```

- As each of the manipulator flags are members of the class ios we need to use the :: scope resolution operator (more on this in a later lecture) to specify the flag as shown below

```
cout.setf( ios::showpos); //set a flag
cout.unsetf(ios::showpos); // unset the flag
```

- We can also specify the precision (number of decimal places) using the setprecision() method as follows

```
cout << setprecision(2) << 10.1234<<endl;
```

A manipulator example

```
#include <iostream>
#include <iomanip>
using namespace std;
void Display(float a[], int width);
int main(void)
{
float Afloat[2]={1.0f,123.287f};
cout << "default format for true "<<true
      << " and false "<<false<<endl;
cout.setf(ios::boolalpha);
cout << "setting boolalpha for true "<<true
      <<" and false "<<false<<endl;
cout <<"Default numeric format"<<endl;
Display(Afloat,10);
cout <<"setting ios::showpoint "<<endl;
cout.setf(ios::showpoint);
Display(Afloat,10);
cout <<"Setting ios::showpos" <<endl;
cout.setf(ios::showpos);
Display(Afloat,10);
cout.unsetf(ios::showpos);
cout <<"setting left justified"<<endl;
cout.setf(ios::left);
cout <<"Setting float precision to 2 decimal places"
      <<endl;
cout.setf(ios::fixed);
cout<<setprecision(2)<<10.5498<<endl;
cout <<"Setting ios::scientific"<<endl;
Display(Afloat,10);
cout.unsetf(ios::scientific);
cout <<"setting ios::hex for value 34"<<endl;
cout.setf(ios::hex,ios::basefield);
cout.setf(ios::showbase);
cout <<34<<endl;
```

```
return 0;
}
void Display(float a[], int width)
{
for(int i=0; i<2; i++)
    {
         cout.width(width);
         cout <<a[i]<<endl;
    }
}
```

- Which gives the following output

```
default format for true 1 and false 0
setting boolalpha for true true and false false
Default numeric format
          1    123.287
setting ios::showpoint
    1.00000    123.287
Setting ios::showpos   +1.00000   +123.287
setting left justified
Setting float precision to 2 decimal places
10.55
Setting ios::scientific
1.00 123.29
setting ios::hex for value 34
0x22
```

Converting Numbers to strings (C Way)

- To convert an integer to a number in C we use the sprintf function as follows

```
#include <iostream>
using namespace std;
int main(void)
{
char Frames[50];
for(int i=0; i<10; i++)
    {
    sprintf(Frames,"TestFrame.%03d.tiff",i);
    cout << Frames<<endl;
    }
return 0;
}
```

How do I convert an integer to a string?

- The simplest way is to use a stringstream:

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
string itos(int i) // convert int to string
{
  stringstream s;
  s << i;
  return s.str();
}
int main()
{
  int i = 127;
  string ss = itos(i);
  const char* p = ss.c_str();
  cout << ss << " " << p << "\n";
}
```

The Frames Thing with C++

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;
int main(void)
{
string Frames;
stringstream  s;

for(int i=0; i<20; i++)
    {
    s.fill('0');
    s.width(3);
    s<<right<<i;
    Frames="TestFrame."+s.str()+".tiff";
    cout << Frames<<endl;
    s.str(""); // clear the string
    }
return 0;
}
```

Using streams to access files

- The streams library also allows access to files using streams there are two basic methods for opening a file either for reading or writing

- These are set with the ios flags as follows

```
ios::out // open a file for writing text
ios::in // open a file for reading text
```

- To open a file we need to declare an object of type *fstream* which is the standard file stream object.

- All operation on the file then reference this object.

- When using files we use the following procedure
  - Create a file object
  - attempt to open the file (if it is not valid exit the program or issue a warning)
  - Loop until file read or write is done
  - close the file

Reading a Text file and printing contents

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
if (argc <=1)
  {
  cout <<"Usage FileRead [filename] "<<endl;
  exit(0);
  }
fstream FileIn;
FileIn.open(argv[1],ios::in);
if (!FileIn.is_open())
    {
    cout <<"File : "<<argv[1]<<" Not found"<<endl;
    exit(1);
    }
string LineBuffer;
unsigned long int LineNumber=1;
while(!FileIn.eof())
   {
    getline(FileIn,LineBuffer,'\n');
    cout <<LineNumber<<" : "<<LineBuffer<<endl;
    LineNumber++;
   }
FileIn.close();
return 0;
}
```

Writing to a File

- To write to a file we use the ios::out flag as shown in the following example

```
#include <iostream>
#include <fstream>
using namespace std;
int main(int argc, char *argv[])
{
if (argc <=1)
     {
      cout <<"Usage FileWrite [filename] "<<endl;
      exit(0);
      }
fstream FileOut;
FileOut.open(argv[1],ios::out);
if (!FileOut.is_open())
     {
      cout <<"Could not open File : "<<argv[1]
           <<" for writing "<<endl;
      exit(1);
      }
for (int i=0; i<10; i++)
     FileOut << "Line number is "<<i<<endl;

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

Binary file reading

- The following example uses binary file I/O to create a user defined number of random points and save them to a file.

- The program then reads them back into a different structure and prints them out.

- Random Number generation is done using the following functions

```
const int HALF_RAND = (RAND_MAX / 2);
/*! returns a random number between +/- 1
float RandomNum(void)
{
  int rn;
  rn = rand();
  return ((float)(rn - HALF_RAND) / (float)HALF_RAND);
}
/*! returns a random number between +/- MaxVal
float RandomNum(float MaxVal)
{
  return MaxVal * RandomNum();
}
```

The program

```
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
using namespace std;
// create a structure to hold the a simple 3D point
typedef struct Point
{
float x;
float y;
float z;
};
int main(int argc, char *argv[])
{
// declare the File pointer object
fstream File;
int NumPoints=0;
cout <<"Please enter the number of points to create"<<endl;
cin >> NumPoints;
// array of points to be written to a file
Point *pointsOut = new Point [NumPoints];
// array of points to be read into the file
Point *pointsIn = new Point [NumPoints];
// check we have a command line argument
if (argc <=1)
    {
     cout <<"Usage BinIO [filename] "<<endl;
     exit(0);
     }
// set the random seed so we get different values each
// time we run the program
srand(time(NULL));
// Open the file for writing
File.open(argv[1],ios::out|ios::binary);
```

```
// if it isn't open exit
if (!File.is_open())
     {
      cout <<"Could not open File : "<<argv[1]<<endl;
      exit(1);
      }
// loop through and create a series of random points
for(int i=0; i<NumPoints; i++)
    {
     pointsOut[i].x=RandomNum(20);
     pointsOut[i].y=RandomNum(20);
     pointsOut[i].z=RandomNum(20);
     }
// write the data to the file (in binary)
File.write(reinterpret_cast <char *>(&pointsOut[0])
            ,sizeof(Point)*NumPoints);
// close the file
File.close();
// now open the file to Read from
File.open(argv[1],ios::in|ios::binary);
// check if ok else exit
if (!File.is_open())
    {
     cout <<"Could not open File : "<<argv[1]<<endl;
     exit(1);
     }
File.read(reinterpret_cast <char *>(&pointsIn[0])
           ,sizeof(Point)*NumPoints);
// now print out points
for(int i=0; i<NumPoints; i++)
    {
    cout <<"["<<pointsIn[i].x<<","<<pointsIn[i].y<<","
         <<pointsIn[i].z<<"]"<<endl;
     }
File.close();
return 0;
}
```

reinterpret_cast

- reinterpret_cast casts a pointer to any other type of pointer. It also allows casting from a pointer to an integer type and vice versa.

- This operator can cast pointers between non-related classed. The operation results is a simple binary copy of the value from one pointer to the other. The content pointed does not pass any kind of check nor transformation between types.

- In the case that the copy is performed from a pointer to an integer, the interpretation of its content is system dependent and therefore any implementation is non portable. A pointer casted to an integer large enough to fully contain it can be casted back to a valid pointer.

- reinterpret_cast treats all pointers exactly as traditional type-casting operators do.

- so we could use the following code in the previous example

```
File.read((char *)&pointsIn[0],sizeof(Point)*NumPoints);
```

The obj File format

- Object files define the geometry and other properties for objects which can be easily used within animation packages.

- Object files can be in ASCII format (.obj) or binary format (.mod). For simplicity the ASCII format will be discussed here as it is easier to parse the data and is a good exercise for file and string handling.

- In its current release, the .obj file format supports both polygonal objects and free-form objects such as curves, nurbs etc.

- For simplicity on polygonal models will be discussed in this section.

Data elements

- The following types of data may be included in an .obj file. In this list, the keyword (in parentheses) follows the data type.
  - geometric vertices (v)
  - texture vertices (vt)
  - vertex normals (vn)
  - face (f)
  - group name (g)
  - smoothing group (s)
  - material name (usemtl)
  - material library (mtllib)

- all values are stored on a single line terminated with a \n (new line character)

- For example

```
v       -5.000000       5.000000       0.000000
vt      -5.000000       5.000000       0.000000
vn       0.000000       0.000000       1.000000
```

Reading a line from an obj file

- The following code segment loops through the file reads a line from the obj file and stores it in a string

- The ParseObj function is used to determine what type the line is

```
string LineBuffer;
do
{
  getline(FileIn,LineBuffer,'\n');
  EntryType=ParseObj(&LineBuffer);
  switch(EntryType)
        {
         case VERTEX : Vertices++; break;
        }
  }while(!FileIn.eof());
```

- The ParseObj function uses the find method of the string class to find the first instance of “v “ it returns -1 if the string is not found

```
enum OBJELEMENTS{VERTEX,FACE,NORMAL,TEXCORD,COMMENT
                 ,GROUP,USEMTL,UNKNOWN};
OBJELEMENTS ParseObj(string *buf)
{
if (buf->find("v ") != -1)
    {
     return VERTEX;
    }
else return UNKNOWN;
}
```