Week7 : Input and Output Streams

Scientific Computing

Blair Jamieson

University of Winnipeg

Class 7

Outline

Outline

Introduction to IO classes

- We have already used `cin` and `cout` to do input from the keyboard, and output to a screen in `<iostream>` header.
- Support also exists for wide characters `wchar_t` for languages with more characters than english.
- For these, can use `wcin` and `wcout`
- Also need to know how to do Input and Output (IO) to named files defined in `<fstream>` header.
- Also can use IO operations to process characters in `string` type objects as defined in `<sstream>` header.

IO Library types and headers

Add `w` in front of any of the types below to make it applicable for `wchar_t` rather than regular `char`.

Library types and headers

Header	Type	Description
`iostream`	`istream`	read from a stream
	`ostream`	writes to a stream
	`iostream`	reads and writes a stream
`fstream`	`ifstream`	read from a file
	`ofstream`	writes to a file
	`fstream`	reads and writes a file
`sstream`	`istringstream`	reads from a `string`
	`ostringstream`	write to a `string`
	`stringstream`	read and write a `string`

Relationship among IO types

- We use the >> operator to read data, and it works same regardless of stream type
  - read from the keyboard with `istream`, using `cin` and >> operator.
  - read from a file using an `ifstream` with the >> operator
  - read from a string using an `istringstream` with >> operator
- We use the << operator to write data to a stream:
  - write to screen with `ostream`, using `cout` and << operator.
  - write to a file using an `ofstream` with << operator
  - write to a string using an `ostringstrem` with << operator
- Everything covered in this lesson applies equally to plain streams, file streams and `string` streams (both `char` and `wchar_t` versions)

Inheritance in IO stream objects

- We will learn more about **inheritance** next class
- Inheritance lets us say that a particular `class` is of the same underlying type as another `class`
- We can use an object of an inherited class as if it were an object of the class it inherits from
- `ifstream` and `istringstream` inherit from `istream`
- thus we can use objects of type `ifstream` or `istringstream` as if they were `istream` objects
- We can use objects of these types in the same way we used `cin`
- Eg. we can use `getline` and >> of `ifstream` or `istreamstring`

No copy or assign for IO objects

- Functions doing IO pass and return the stream through references
- Reading or writing an IO object changes its state – so reference must not be const

```
1 ofstream out1, out2;
2 out1 = out2; // error: cannot assign stream objects
3 // below error: can't initialize ofstream parameter
4 ofstream print( ofstream );
5 // below error: cannot copy stream objects
6 out2 = print( out2 );
```

IO Condition states

- Errors can occur when reading in values
- For example if we try to read in an `int` but someone enters a non-`int` value

```
int val;
cin >> val;
```

- If non-`int` is entered, `cin` will be in an error state
- Once an error occurs, subsequent IO operations on stream will fail
- Can only read/write a stream when it is in a non-error state
- Can check state of stream in a condition:

```
while ( cin >> val ){
  // read successful... use val
}
```

Table of condition states

- Stream state is stored as set of bits in *strm*`::iostate`
- For example `cin::eofbit` is set to 1 if reached end-of-file condition when calling `cin`

IO library condition states	
strm`::iostate`	*strm* is one of IO types (eg. ifstream), `iostate` integer representing state
strm`::badbit`	`iostate` for stream corrupted
strm`::failbit`	`iostate` for IO operation failed
strm`::eofbit`	`iostate` for stream at end of file
strm`::goodbit`	`iostate` for no-error (zero)

Table of methods to access condition states

- Several useful functions to access the condition of a stream exist:

IO library methods accessing condition states	
`s.eof()`	`true` if `eofbit` in stream `s` set
`s.fail()`	`true` if `failbit` or `badbit` set
`s.bad()`	`true` if `badbit` of `s` set
`s.good()`	`true` if `s` in valid state
`s.clear()`	`true` if `badbit` of `s` set
`s.clear(flags)`	reset condition value to valid state
`s.setstate(flags)`	reset condition of `flags`
`s.rdstate()`	returns *strm*`::iostate`

Checking the stream state

- Stream state is named `iostate` that is a collection of bits to indicate particular kinds of IO conditions
- `badbit` indicates system-level failure, such as unrecoverable read or write – usually not possible to use stream once `badbit` is set
- `failbit` indicates recoverable error – such as reading a character when a numeric value expected
- `eofbit` indicates end-of-file – if eof doesn't follow good input `failbit` also set
- `goodbit` guaranteed to have value 0 if no failures on stream
- Use `good()` and `bad()` to determine overall state of stream.

Managing the condition state

- `rdstate` member returns `iostate` value
- `setstate` turns on the given condition bit(s) to indicate that a problem occurred
- `clear()` with no arguments clears all failure bits
- Example using stream state:

```
auto savestate = cin.rdstate();
cin.clear(); // clears failure bits
process_input( cin ); // use cin in a function
// below: reset cin to its prev state
cin.setstate( savestate );
// To reset failbit, badbit (leave eofbit alone):
if ( cin.fail() && cin.badbit() ){
  cin.clear( cin.rdstate() & ~cin.failbit &
      ~cin.badbit );
}
```

Managing the output buffer

- Each output stream manages a buffer which holds the data the program reads and writes
- Writing to stream may occur immediately, or be stored in a buffer to be output later
- Reason for storing in a buffer is that IO tends to be time-consuming – so do it in larger blocks when possible
- Conditions that cause the buffer to be flushed are:
  - program completes normally
  - buffer becomes full
  - we flush the buffer with `endl` or `flush`
  - we set `unitbuf` so that stream empty's buffer after each operation: `cerr` is set that way
  - another stream tied to same output may fill buffer: `cerr` and `cout` both tied to screen

Flushing output buffer

- `endl` flushes buffer (and writes a newline)
- `flush` flushes buffer, without writing newline

```
// below writes hi, newline and flush buffer
cout << "hi!" << endl;
// below writes hi and flushes buffer
cout << "hi!" << flush;
// below writes hi and a null then flushes buffer
cout << "hi!" << ends;
```

- To flush after every output:

```
cout << unitbuf; // all writes to flush immediately
cout << nounitbuf; // return to normal buffering
```

Caution about output buffer flush

- Output buffers are not flushed if the program terminates abnormally
- If program is crashing add `try, catch` blocks and `cout << flush` to get any buffered output before terminating

Tying IO streams together

- `cin` is tied to `cout` – when `cin >> ival` is called, the `cout` buffer is flushed
- Doing so makes sure all output including prompts to the user will be written before attempting to read the input
- Two overloaded versions of `tie`
  - One takes no arguments and returns a pointer to output stream, if any, to which object is tied (or `nullptr`)
  - Second version takes pointer to `ostream` and ties itself to that stream

```
1 // * illustration only *
2 // library already ties these
3 cin.tie( &cout );
4 // below cin no longer tied
5 ostream *old_tie = cin.tie( nullptr );
6 // below ties cin and cerr -- not a good idea
7 cin.tie( &cerr );
8 //reestablish normal tie b/w cin and cout
9 cin.tie( old_tie );
```

`fstream` file IO

- `ifstream` for input from file and `ofstream` for output
- Use same << and >> operators on them as `cin` and `cout`

fstream `fstrm`	creates filestream named `fstrm` *fstream* is ifstream, ofstream,...
fstream `fstrm(s)`	creates *fstream* and opens file named `s` as `string` or c-char array
fsteam `fstrm(s,mode)`	opens file in particular mode `in, out, app, ...`
`fstrm.open(s)`	open file `s`, binds it to `fstrm`
`fstrm.open(s,mode)`	open file `s`, in mode
`fstrm.close()`	close file
`fstrm.is_open()`	returns `true` if file is open

Using file stream objects

- To read or write a file, we define a file stream object and associate a file with it:

```
1 // construct ifstream, open file named "filename"
2 // in is now the name of the ifstream object
3 // below: c-style char array used for filename
4 ifstream in( "filename" );
5 // define output file stream: no file yet
6 ofstream out;
7 string fname { "somefile.txt" };
8 // since c++11 can use string for filename
9 ifstream in2( fname );
```

Using `fstream` in place of `iostream`

- We can use an object of an inherited type in place of type expected
- Functions expecting reference (or pointer) to **`istream`** can also be provided with **`ifstream`** or **`istringstream`**
- Consider defining **`readDate`** function:

```
istream &readDate( istream &is, Date &d ){
  int ad, am, ay; char c;
  if ( is >> ay >> '/' >> am >> '/' >> ad ){
    d = Date{ ay, Date::Month(am), ad };
  }
  return is;
}
```

- We can call **`readDate`** with **`ifstream`**, or **`istringstream`**:

```
istringstream adate{"2012 11 04"};
ifstream in("fileofdates.txt");
Date dfromfile, dfromstring;
if ( readDate( in, dfromfile ) )
  cout<<"Date in file:" << dfromfile << endl;
if ( readDate( adate, dfromstring ) )
  cout<<"Date in sstream:" << dfromstring << endl;
```

Example `fileIO.cpp`

- We can try reading and writing dates from file, string-streams and cin/cout.
- Add another function to write a date to `ostream` type objects:

```
1 // writeDate takes output stream and writes a date
2 ostream &writeDate( ostream &os, const Date &d ){
3   os << d.year() << '/' << d.month() << '/'
4      << d.day() << ' ';
5   return os;
6 }
```

- See examples in `fileIO.cpp`

Opening and closing files

- If we define empty file stream, can call `open` method to associate file with stream
- Need to check if `open` succeeded by testing state of stream
- if `open` fails, `failbit` is set on the stream

```
ifstream fin("infile.txt");
ifstream fout;
string fname{"outfile"};
fout.open( fname + ".txt");
if ( fout ) { /* ... */ }
if ( fin ) { /* ... */ }
// can also check if file is open
if ( fin.is_open() ){ /* ... * / }
// can close stream and open it to another file
fin.close();
fin.open("infile2.txt");
```

Automatic construction and close

- Suppose we have `main` function that takes list of files to process as command line arguments
- Loop through files might look like:

```
1 for ( auto iter = argv+1 ; iter != argv+argc ;
      ++iter ){
2   ifstream fin( *iter );
3   if ( fin ){
4     process_file( fin );
5   } else {
6     cerr << "couldn't open : "<< *iter << endl;
7   }
8 }
```

- Note each iteration makes its own `ifstream` object which gets destroyed at end of each loop
- Automatically closed when ifstream object goes out of scope

File modes

- default `ifstream` are opened in `in`
- default `ofstream` are opened in `out` and `trunc`

mode	meaning
`in`	open for input
`out`	open for output
`app`	open to append to end of file
`ate`	open and seek to end of file
`trunc`	truncate (empty) the file
`binary`	do IO operations in binary

More about file modes

- Opening file by default in `out` mode discards existing contents of file
- To preserve file contents open in `ofstream::app` mode
- File mode of stream can change at each `open`

```
// below: out and trunc implicit
ofstream out1("preciousdata");
// below: trunc implicit
ofstream out2("file2", ofstream::out );
ofstream out3("file3", ofstream::out |
    ofstream::trunc );
// below: append to file (out implied)
ofstream out4("file4", ofstream::app);
out1.close();
// below: reopen stream to append rather than
    truncate
out1.open("preciousdata", ofstream::app);
```

introduction to `string` streams

- `sstream` header defines three IO types to read/write from strings in memory
  1. `istringstream` reads a `string`
  2. `ostringstream` writes a `string`
  3. `stringstream` reads and writes a `string`
- Operations on `stringstream` objects:

sstream `strm;`	`strm` is an unbound `stringstream`, *sstream* is one of types in `sstream`
sstream `strm(s);`	`stringstream` bound to string `s`
`strm.str()`	returns copy of `string` in `strm`
`strmstr(s)`	copies string in `s` to `strm`

- Note that these operations are unique to `sstream` types, and not for `fstream`
- similarly `open` and `close` of `fstream` can't be used on `sstream` objects

using `istringstream`

- `istringstream` often used for parsing lines of a file
- Example file of monthly average weather: with station name, mean temp, max temp, min temp, mm of precip, days with precip:

```
SPRAGUE             13.8 25.7 0.2 96 7
STONYMOUNTAIN  13.8 23.5 1 26.8 6
WINNIPEGTHEFORKS 15.1 27.4 4.7 25.2 5
CARMANUOFMCS  14.1 28.4 0.5 64.7 7
WINNIPEGACS  14.4 27.7 0.1 35.3 5
BERENSRIVERCS  12.4 24.6 -1.3 91.1 10
```

- We can define a structure `WeatherInfo` to the values into

```
1 struct WeatherInfo{
2   string station;
3   float  T_mean;
4   float  T_max;
5   float  T_min;
6   float  Precip_mm;
7   float  Precip_days;
8 };
```

Example using `istringstream`

- Example in `weatherSS.cpp`:

```
1  istream & readWeather( istream& is, WeatherInfo &w
       ){
2    is >> w.station >> w.T_mean >> w.T_max >> w.T_min
3       >> w.Precip_mm >> w.Precip_days;
4    return is;
5  }
6  //...
7  ifstream fin("weather.txt");
8  string line, word;
9  vector< WeatherInfo > weather;
10 while ( fin && getline( fin, line ) ){
11   WeatherInfo wi;
12   istringstream iss ( line );
13   if ( readWeather( iss, wi ) )
14     weather.push_back( wi );
15 }
```

- `weather` vector now holds all of the records

Using ostringstream

- `ostringstream` is useful for building up a string of data that could eventually be output to file or screen
- For example we may want to check temperatures and build up list of good and bad readings

```
1  void checkRecord( ostream & good, ostream & bad,
2                    const WeatherInfo & w ){
3    if ( isValidWeather( w ) ) printWeather( good );
4    else printWeather( bad );
5    }
6  }
7  //...
8  ostringstream formatted, badNums;
9  for ( const auto &rec : weather ){
10   checkRecord( formatted, badnums, rec )
11 }
12 // print good records to file
13 ofstream fgood("good.txt"), fbad("bad.txt");
14 if ( fgood ) fgood << formatted ;
15 if ( fbad ) fbad << badNums ;
```

Examples in `weatherSS.cpp`

- Have a look at the examples using `ifstream`, `ofstream`, `istringstream`, `ostringstream` in the file `weatherSS.cpp`

Outline

Formatted IO with streams

- Definition of **manipulators** that change the format that will be used for a stream are defined in `iomanip` header file
- We have already used manipulator `endl` that writes newline and flushes buffer
- Manipulators return stream object to which it was applied, so can be used in subsequent stream operations in same statement
- Manipulators change format state of stream for all subsequent IO
- Best to put back to defaults after use
- Example: format of boolean values

```
1 cout << "default bool values : "
2      << true << ", " << false << endl;
3 cout << "alpha bool values : "
4      << boolalpha << true << ", " << false << endl;
```

- Above prints:

```
default bool values : 1 0
alpha bol values : true false
```

Resetting manipulator state

- Example of printing bool value as alpha then returning to default state:

```
bool bval = isValid( somedate );
cout << "Date valid? " << boolalpha << bval
     << noboolalpha << endl;
```

- Example specifying base (only for integer) value output:

```
cout << "default: " << 20 << " " << 1024 <<endl;
cout << "octal: " << oct << 20 << " " << 1024
   <<endl;
cout << "hex: " << hex << 20 << " " << 1024 <<endl;
cout << "decimal: " << dec << 20 << " " << 1024
   <<endl;
```

- above prints:

```
default: 20 1024
octal: 24 2000
hex: 14 400
decimal: 20 1024
```

Showing base on output

- Showing base prints 0x to indicate hexadecimal 0 to indicate decimal
- Using same example as before, can use `showbase` manipulator:

```
1 cout << showbase;
2 cout << "default: " << 20 << " " << 1024 <<endl;
3 cout << "octal: " << oct< << 20 << " " << 1024
    <<endl;
4 cout << "hex: " << hex << 20 << " " << 1024 <<endl;
5 cout << "decimal: " << dec << 20 << " " << 1024
    <<endl;
6 cout << noshowbase;
```

- Above prints:

```
default: 20 1024
octal: 024 02000
hex: 0x14 0x400
decimal: 20 1024
```

- Can also use `uppercase` or `nouppercase` to change between

Formatting float values

- Three aspects to control:
  1. Number of digits of precision
  2. Whether number is printed as hex, fixed decimal or scientific notation
  3. Whether decimal point is printed for whole numbers
- `precision()` returns the current precision for float values output on the stream
- `setprecision( value )` is used to set the number of significant digits to print
- Default is 6 digits of precision.

Examples of formatting float values

- Various ways of printing float values:

```
1 // print current precision then print pi
2 cout << "Precision "<< cout.precision()
3      << " pi=" << acos(-1) << endl;
4 cout.setprecision(12); // use method of cout
5 cout << "Precision " << cout.precision()
6      << " pi=" << acos(-1) << endl;
7 cout << setprecision(3); // use manipulator
8 cout << "Precision " << cout.precision()
9      << " pi=" << acos(-1) << endl;
```

- Output from above will be:

```
Precision 6 pi=3.14159
Precision 12 pi=3.14159265359
Precision 3 pi=3.14
```

Specifying notation for float numbers

- Options for notation are: `defaultfloat`, `scientific`, `fixed` and `hexfloat`

```
cout <<"default: "<<100*sqrt(2.0)<<endl;
cout <<"sci: "<<scientific <<100*sqrt(2.0)<< endl;
cout <<"fixed: " <<fixed <<100*sqrt(2.0)<<endl;
cout <<"hex: " <<hexfloat <<100*sqrt(2.0)<<endl;
cout <<"default: " << defaultfloat
     << 100*sqrt(2.0) << endl;
```

- Above prints:

```
default: 141.421
sci: 1.41214e+002
fixed: 141.421356
hex: 0x1.1ad7bcp+7
default: 141.421
```

- `hexfloat` and `defaultfloat` not implemented in the MinGW compiler we are using
- To reset to default we can use:

```
cout.unsetf(std::ios_base::floatfield);
```

Printing the decimal point

- When fractional part of decimal is zero, default is not to print a decimal place
- Can force printing or not printing decimals with manipulators `showpoint` and `noshowpoint`

```
1 cout << 42.0 << endl; //prints 42
2 // below will print 42.0000
3 cout << showpoint << 42.0 << endl;
4 cout << noshowpoint; // revert to default
```

Padding the output

- When printing a table of numbers, often want to have columns line up, to help can use formatting:
  - `setw` to specifies minimum space for *next* numeric value
  - `left` to left-justify, and `right` to right-justify
  - `internal` to control placement of sign on neg. values – `internal` left justifies sign, and right justifies number
  - `setfill` specifies character to pad output (default is space)

```
1  void coutid(const int &i, const double &d){
2    cout << "i|" << setw(12) << i <<"|"<<endl;
3    cout << "d|" << setw(12) << d <<"|"<<endl;
4  } //...
5  int i = -16; double d = acos(-1);
6  coutid(i,d); // print with width 12:
7  cout << left; // left justify
8  coutid(i,d);
9  cout << right; // restore right justify
10 cout << internal; // pad number rt. just -
11 coutid(i,d);
12 cout << setfill('0'); // fill with zeros
13 coutid(i,d);
14 cout << setfill(' '); // return to default
```

Result of padding the output

- Result of padding code on previous page is:

```
i|          -16|
d|     3.14159|
i|-16          |
d|3.14159      |
i|-          16|
d|     3.14159|
i|-00000000016|
d|000003.14159|
```

Input formatting?

- Default is to ignore whitespace on input
- Can override this by setting manipulator **`noskipws`** and returning to default with **`skipws`** (skip white space)
- For example given the sequence:

```
a   b c
d     e
```

- And input code:

```
1 char ch;
2 while ( cin >> ch ) cout << ch;
3 cout <<endl;
4 cin >> noskipws; //set to read whitespace
5 while ( cin >> ch ) cout << ch;
6 cin >> skipws; //set to ignore whitespace
```

- Result is:

```
abcde
a   b c
d     e
```

Outline

Example problem statement

- Using what we have learned until now, we can solve a fairly large set of problems
- One limitation is that we are doing screen based problems, so any display of results has to be numeric or character based
- Lets suppose we want to make a class (or set of classes)that can represent a set of data, and plot it to the terminal
- Data will be columns of data: x y error_bar_y
- May get data from a vector, array or from cin
- Clearly it would be better to do this with a graphics window – but suppose we want to do this using a set of characters...
- How might you solve this problem?

Designing the Graph class

- Suppose we define the graph object as a class in the namespace `SC` where we can put any helper functions
- First we can design the interface for the Graph class
- Lets call the main graph class `SC::Graph` and we can initialize it in one of following ways:

```
SC::Graph gr1( title, xlabel, ylabel );  // empty graph
// below vecx, vecy and vecdy are of vector<double> type.
SC::Graph gr2( title, xlabel, ylabel,
               vecx, vecy, vecdy );
// below arrx, arry and arrdy are arrays of double
// of length n, ie
const int n = 100; // number of data points
double arrx[n], arry[n], arrdy[n];
SC::Graph gr3( title, xlabel, ylabel,
                n, arrx, arry, arrdy );
```

Graph class constructors

- Graph class definition so far:

```
1  typedef std::string SS;
2  typedef std::vector<double> VD;
3  class Graph{
4    public:
5      Graph(); // default constructor
6      Graph( const SS & atitle, const SS & axlabel,
           const SS & aylabel );
7      Graph( const SS & atitle, const SS & axlabel,
           const SS & aylabel,
8              const VD &ax, const VD &ay, const VD
                 &ady );
9      Graph( const SS & atitle, const SS & axlabel,
           const SS & aylabel,
10             const unsigned an, const double ax[],
                 const double ay[],
11             const double ady[] );
```

Designing the Graph class – methods

- Need methods to add data (or change data set) particularly for case of empty graph
- Methods we could add for this might look like following, along with constructors:

```
1  class Graph {
2    public:
3      //...
4      // Methods for adding data to a graph:
5      // ask for input by Cin
6      void DataCin();
7      // set data with SDvd types
8      void SetData( const VD vecx, const VD vecy,
           const VD vecdy );
9      // set data with arrays
10     void SetData( const int n, const double ax[],
           ay[], ady[] );
11     // ...
12 };
```

Graph Methods for accessing axes

- Lets define a separate `class` called `SC::Axis` to hold the axis information and that knows about how to draw the axis, and find where on the axis a particular value falls

```
1  class Graph {
2    public:
3      // ...
4      void SetAxes( const Axis & axx, const Axis &
           axy );
5      void SetXaxis( const Axis & axx );
6      void SetYaxis( const Axis & axy );
7      void GetAxes( Axis & xax, Axis & yax );
8    private:
9      Axis fXax;
10     Axis fYax;
11     //...
12 };
```

Graph Methods for Drawing the graph

- Well... we have no canvas to draw on, but we have a terminal with so many characters wide and so many characters high
- So lets assume we define a vector of strings vector< string >, where each entry in the vector is the number of characters wide to match the width of the terminal, and the number of entries in the vector is the height of the terminal
- Assume default terminal is 80 wide, and 30 high

```
1  class Graph {
2    public:
3      //...
4      void Draw( std::vector< SS > &aScreen );
5      void Print();
6  };
7  // also add helper function to clear the "screen"
8  void GraphScreenClear(
9    std::vector< std::string> & aScreen,
10   const unsigned ax=80, const unsigned ay=30 );
```

Private data hidden in the graph object

- Graph need to hold the title for the graph, x-axis data points, y-axis data, information about the axes and a marker type for the points on the graph:

```
typedef std::vector<double> VD;
typedef std::string SS;
class Graph {
  //...
  private:
    SS fTitle;
    VD fX;
    VD fY;
    VD fdY;
    Axis fXax;
    Axis fYax;
    char fMarker;
};
```

Designing the interface to the `Axis` class

- `SC::Axis` holds axis label, range, number of divisions, and orientation
- We will draw the axis differently if it is vertical or horizontal
- Axis initialization only in one of two ways:

```
1 class Axis {
2   public:
3     Axis(); // default constructor
4     Axis( const std::string &alabel, const double
          axmin,
5           const double axmax, const bool
              ishorizontal );
6 //...
7 };
```

- Default will have 5 divisions on axes

Axis methods for getting / setting axis

- We need to be able to change the axis range, change the axis label, change the number of divisions, change the orientation of the axis, axis range, and methods to get these values.

```
class Axis {
  public:
  //...
  void SetRange( axmin, axmax );
  void SetLabel( alabel );
  void SetNDivs( an ); // number of axis ticks
  void SetHorizontal(); // make axis x-axis
  void SetVertical(); // make axis y-axis
  void GetRange( axmin, axmax ) const;
  void GetLabel( alabel ) const;
  int GetNDivs() const;
// ...
};
```

Other Axis methods – about the screen

- We will let the Axis class decide where to put things on the screen
- Needs to know about screen size to place the axis and get where on the screen a point should fall
- For vertical axes lets use 10th column of characters on screen for the axis, first nine columns can hold numbers along the axis
- First row of screen will be for the title
- Second row for the y-axis label
- third to last row will be x-axis, 2nd to last row will be x-axis numbers, last row will be x-axis label

```
1 class Axis{
2   public: // ...
3     unsigned GetLoc( vector< string >& aScreen,
4         const double aval ) const;
5     void Draw( vector< string > & aScreen ) const;
6   private:
7     // ... Helper methods for drawing X or Y axis
8     void DrawX( vector< string > & aScreen );
9     void DrawY( vector< string > & aScreen );
```

Implementing all the methods

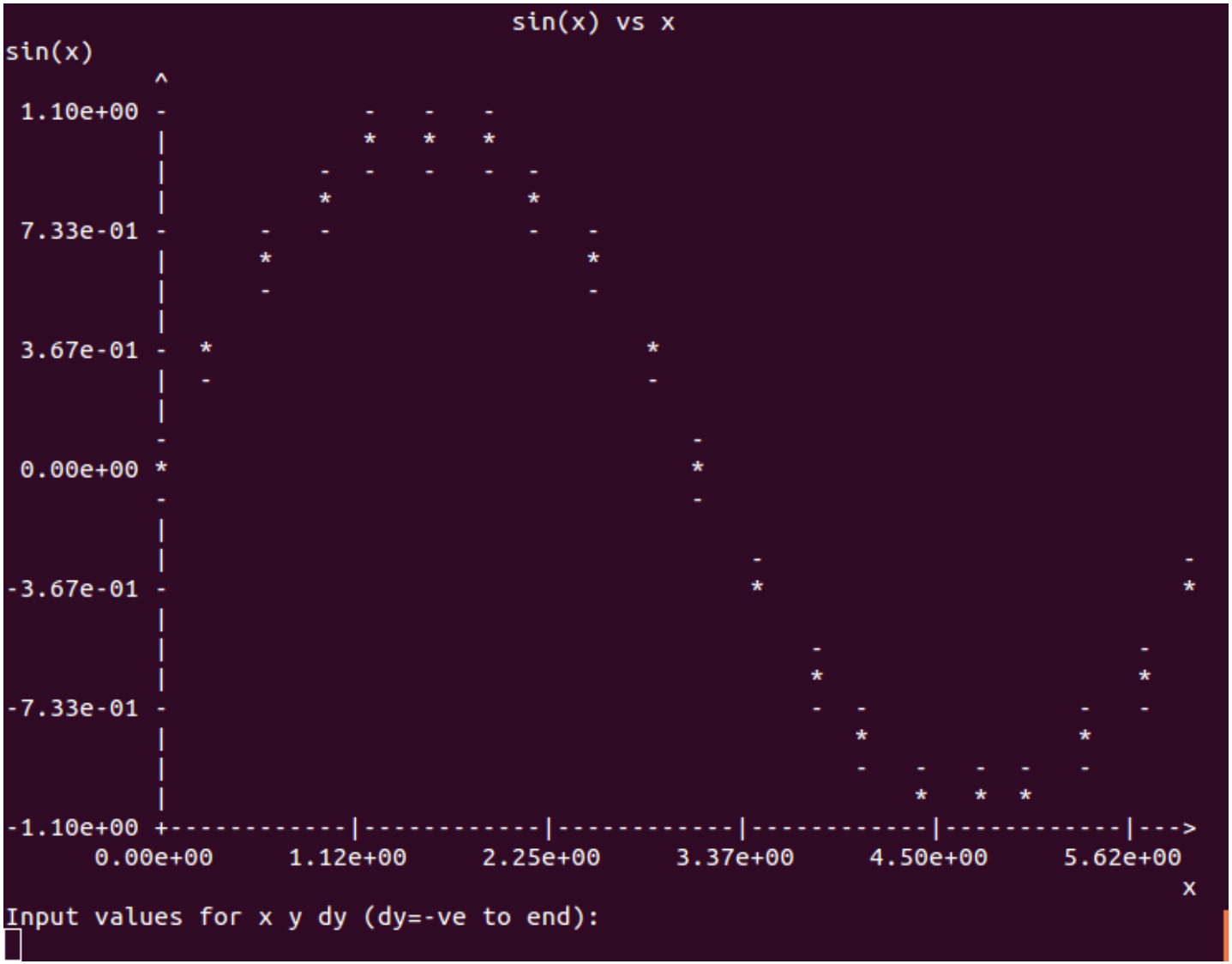
- While the interface in place took a while to lay out..
- It will also take a fair bit of work to get the implementation of all the class methods working
- Perhaps the hardest ones are Drawing the Axes, and implementing the `GetLoc` method of the axes that tell us where to put each datapoint
- In fact, not convinced I have that code right yet, but it does something close to right
- Have a look at the files `SCAxis.h` and `SCGraph.h` that hold the full interface and helper functions declared
- Have a look at `SCAxis.cpp` and `SCGraph.cpp` that have the implementation of the methods
- You will see that while the code is a bit long, there is no piece of code there that is not similar to what we have already learned!

Testing the Graph class

- Lets plot $\sin x$ vs x for 20 points from 0 to 2π, with an arbitrary error bar
- Code for that is now fairly short:

```
int main(){
  const int npts = 20;
  std::vector<double> xx, yy, ee;
  double dx = 2.*std::acos(-1)/npts;
  for ( int i = 0; i< npts; i++){
    xx.push_back( dx * i );
    yy.push_back( std::sin( dx * i ) );
    ee.push_back(
     std::fabs(0.05*std::sin( dx*i ))+0.05 );
  }
  Graph g1( "sin(x) vs x", "x",
            "sin(x)", xx, yy, ee );
  std::vector< std::string > ascreen;
  GraphScreenClear( ascreen, 80, 30 );
  g1.Draw( ascreen );
```

Output from `testGraph.exe`



Week7 Done!

- Homework #5 due Fri Oct. 28 (17:00).
- Reminder – midterm project due today Oct. 25 (17:00).
- Final project given out today is due Nov. 29 (17:00)