Chapter 11. I/O classes

Authors: *Joseph T. Buck*

Other Contributors: *Bilung Lee*

11.1 StringList, a kind of String class

Class `StringList` provides a mechanism for organizing a list of strings. It can also be used to construct strings of unbounded size, but the class `InfString` is preferred for this. It is privately derived from `SequentialList`. Its internal implementation is as a list of `char*` strings, each on the heap. A `StringList` object can be treated either as a single string or as a list of strings; the individual substrings retain their separate identity until the conversion operator to type `const char*` is invoked. There are also operators that add numeric values to the `StringList`; there is only one format available for such additions. WARNING: if a function or expression returns a `StringList`, and that value is not assigned to a `StringList` variable or reference, and the `(const char*)` cast is used, it is possible (likely under g++) that the `StringList` temporary will be destroyed too soon, leaving the `const char*` pointer pointing to garbage. Always assign a temporary `StringList` to a `StringList` variable or reference before using the `const` char* conversion. Thus, instead of

```
function_name(xxx,(const char*)functionReturningStringList(),yyy);
```

one should use

```
StringList temp_name = (const char*)functionReturningStringList();
function_name(xxx,temp_name,yyy);
```

This includes code like

```
strcpy(destBuf,functionReturningStringList());
```

which uses the `const char*` conversion implicitly.

11.1.1 StringList constructors and assignment operators

The default constructor makes an empty `StringList`. There is also a copy constructor and five single-argument constructors that can function as conversions from other types to type `StringList`; they take arguments of the types `char`, `const char *`, `int`, `double`, and `unsigned int`. There are also six assignment operators corresponding to these constructors: one that takes a `const StringList&` argument and also one for each of the five standard types: `char`, `const char *`, `int`, `double`, and `unsigned int`. The resulting object has one piece, unless initialized from another `StringList` in which case it has the same number of pieces.

11.1.2 Adding to StringLists

There are six functions that can add a printed representation of an argument to a `StringList`: one each for arguments of type `const StringList&`, `char`, `const char *`, `int`, `dou-`

`ble`, and `unsigned int`. In each case, the function can be accessed in either of two equivalent ways:

```
StringList& operator += (type arg);
StringList& operator << (type arg);
```

The second "stream form" is considered preferable; the "+=" form is there for backward compatibility. If a `StringList` object is added, each piece of the added `StringList` is added separately (boundaries between pieces are preserved); for the other five forms, a single piece is added.

11.1.3 StringList information functions

```
const char* head() const;
```

Return the first substring on the list (the first "piece"). A null pointer is returned if there are none.

```
int length() const;
```

Return the length in characters.

```
int numPieces() const;
```

Return the number of substrings in the `StringList`.

11.1.4 StringList conversion to const char *

```
operator const char* ();
```

This function joins all the substrings in the `StringList` into a single piece, so that afterwards `numPieces` will return 1. A null pointer is always returned if there are no characters. Warning: if this function is called on a temporary `StringList`, it is possible that the compiler will delete the `StringList` object before the last use of the returned `const char *` pointer. The result is that the pointer may wind up pointing to garbage. The best way to work around such problems is to make sure that any `StringList` object "has a name" before this conversion is applied to it; e.g., assign the results of functions returning `StringList` objects to local `StringList` variables or references before trying to convert them.

```
char* newCopy() const;
```

This function makes a copy of the `StringList`'s text in a single piece as a `char*` in dynamic memory. The object itself is not modified. The caller is responsible for deletion of the returned text.

11.1.5 StringList destruction and zeroing

```
void initialize();
```

This function deallocates all pieces of the `StringList` and changes it to an empty `StringList`.

```
~StringList();
```

The destructor calls the `initialize` function.

11.1.6 Class StringListIter

Class `StringListIter` is a standard iterator that operates on `StringLists`. Its `next()` function returns a pointer of type `const char*` to the next substring of the `StringList`. It is important to know that the operation of converting a `StringList` to a `const char*` string joins all the substrings into a single string, so that operation should be avoided if extensive use of `StringListIter` is planned.

11.2 InfString, a class supporting unbounded strings

Class `InfString` provides a mechanism for building strings of unbounded size. It provides a subset of the functions in a typical C++ String class. Strings can be built up piece by piece. As segments are added, they are copied, so the caller need not keep the segments around. Upon casting to `(char*),` the strings are collapsed into one continuous string, and a pointer to that string is returned. The calling function can treat this as an ordinary pointer to an ordinary array of characters, and can modify the characters. But the length of the string should not be changed, nor should the string be deleted. The `InfString` destructor is responsible for freeing the allocated memory. `InfString` is publically derived from `StringList`, adding only the cast `char*`. Its internal implementation is as a list of `char*` strings, each on the heap. The individual substrings retain their separate identity until the conversion cast to type `char*` is invoked, although if access to the individual strings is needed, then `StringList` should be used. There are also operators that add numeric values to the `StringList`; there is only one format available for each such addition. WARNING: if a function or expression returns an `InfString`, and that value is not assigned to an `InfString` variable or reference, and the `(char*)` cast is used, it is possible (likely under g++) that the `InfString` temporary will be destroyed too soon, leaving the `char*` pointer pointing to garbage. Always assign temporary `InfString` to `InfString` variables or references before using the `char*` conversion. Thus, instead of

```
function_name(xxx,(char*)functionReturningInfString(),yyy);
```

one should use

```
InfString temp_name = (char*)functionReturningInfString();
function_name(xxx,temp_name,yyy);
```

This includes code like

```
strcpy(destBuf,functionReturningInfString());
```

which uses the `char*` conversion implicitly.

11.2.1 InfString constructors and assignment operators

The default constructor makes an empty `InfString`. There is also a copy constructor and six single-argument constructors that can function as conversions from other types to type `InfString`; they take arguments of the types `char`, `const char*`, `int`, `double`, `unsigned int`, and `const StringList&`. There are also seven assignment operators corresponding to these constructors: one that takes a `const InfString&` argument and also one for each of the six standard types: `char`, `const char*`, `int`, `double`, `unsigned int`, and `const StringList&`.

11.2.2 Adding to InfStrings

There are seven functions that can add a printed representation of an argument to a InfString: one each for arguments of type `const InfString&`, `char`, `const char*`, `int`, `double`, `unsigned int`, and `const StringList&`. In each case, the function can be accessed in either of two equivalent ways:

```
InfString& operator += (type arg);
InfString& operator << (type arg);
```

The second "stream form" is considered preferable; the "+=" form is there for backward compatibility. If a `InfString` object is added, each piece of the added `InfString` is added separately (boundaries between pieces are preserved); for the other five forms, a single piece is added.

11.2.3 InfString information functions

```
int length() const;
```

Return the length in characters.

11.2.4 InfString conversion to char *

```
operator char* ();
```

This function joins all the substrings in the `InfString` into a single piece, a returns a pointer to the resulting string. A null pointer is always returned if there are no characters. Warning: as pointed out above, if this function is called on a temporary `InfString`, it is possible that the compiler will delete the `InfString` object before the last use of the returned `char*` pointer. The result is that the pointer may wind up pointing to garbage. The best work-around for such problems is to make sure that any `InfString` object "has a name" before this conversion is applied to it; e.g. assign the results of functions returning `InfString` objects to local `InfString` variables or references before trying to convert them.

```
char* newCopy() const;
```

This function makes a copy of the `InfString`'s text in a single piece as a `char*` in dynamic memory. The `InfString` object itself is not modified. This is useful when the caller wishes to be responsible for deletion of the returned text.

11.2.5 InfString destruction and zeroing

```
void initialize();
```

This function deallocates all pieces of the `InfString` and changes it to an empty `InfString`.

```
~InfString();
```

The destructor calls the `initialize` function.

11.2.6 Class InfStringIter

Class `InfStringIter` is a standard iterator that operates on `InfStrings`. However, the `InfString` class is not intended for use when access to the individual components of the string is desired. Use `StringList` for this.

11.3 Tokenizer, a simple lexical analyzer class

The `Tokenizer` class is designed to accept input for a string or file and break it up into tokens. It is similar to the standard istream class in this regard, but it has some additional facilities. It permits character classes to be defined to specify that certain characters are white space and others are “special” and should be returned as single-character tokens; it permits quoted strings to override this, and it has a file inclusion facility. In short, it is a simple, reconfigurable lexical analyzer. `Tokenizer` has a public const data member named `defWhite` that contains the default white space characters: space, newline, and tab. It is possible to change the definition of white space for a particular constructor.

11.3.1 Initializing Tokenizer objects

`Tokenizer` provides three different constructors:

```
Tokenizer();
```
The default constructor creates a `Tokenizer` that reads from the standard input stream, `cin`. Its special characters are simply \key (and \key).

```
Tokenizer(istream& input,const char* spec,
        const char* w = defWhite);
```
This constructor creates a `Tokenizer` that reads from the stream named by *`input`*. The other arguments specify the special characters and the white space characters.

```
Tokenizer(const char* buffer,const char* spec,
        const char* w = defWhite);
```
This constructor creates a `Tokenizer` that reads from the null-terminated string in *`buffer`*. `Tokenizer`’s destructor closes any include files associated with the constructor and deletes associated internal storage. The following operations change the definition of white space and of special characters, respectively:

```
const char* setWhite(const char* w);
const char* setSpecial(const char* s);
```
In each case, the old value is returned. By default, the line comment character for `Tokenizer` is #. It can be changed by

```
char setCommentChar(char n);
```
Use an argument of 0 to disable the feature. The old comment character is returned.

11.3.2 Reading from Tokenizers

The next operation is the basic mechanism for reading tokens from the `Tokenizer`:

```
Tokenizer& operator >> (char* pBuffer);
```
Here *`pBuffer`* points to a character buffer that reads the token. There is a design flaw: there isn’t a way to give a maximum buffer length, so overflow is a risk. By analogy with streams, the following operation is provided:

```
operator void*();
```
It returns null if `EOF` has already been reached and non-null otherwise. This permits loops

like

```
Tokenizer tin;
while (tin) { ... do stuff ... }
int eof() const;
```

Returns true if the end of file or end of input has been reached on the `Tokenizer`. It is possible that there is nothing left in the input but write space, so in many situations `skipwhite` should be called before making this test.

```
void skipwhite();
```

Skip white space in the input.

```
void flush();
```

If in an include file, the file is closed. If at the top level, discard the rest of the current line.

11.3.3 Tokenizer include files

`Tokenizer` can use include files, and can nest them to any depth. It maintains a stack of include files, and as `EOF` is reached in each file, it is closed and popped off of the stack. The method

`int fromFile(const char*` *name*);

opens a new file and the `Tokenizer` will then read from that. When that file ends, `Tokenizer` will continue reading from the current point in the current file.

```
const char* current_file() const;
int current_line() const;
```

These methods report on the file name and line number where `Tokenizer` is currently reading from. This information is maintained for include files. At the top level, `current_file` returns a null pointer, but `current_line` returns one more than the number of line feeds seen so far.

```
int readingFromFile() const;
```

Returns true (1) if the `Tokenizer` is reading from an include file, false (0) if not.

11.4 pt_ifstream and pt_ofstream: augmented fstream classes

The classes `pt_ifstream` and `pt_ofstream` are derived from the standard stream classes `ifstream` and `ofstream`, respectively. They are defined in the header file `pt_fstream.h`. They add the following features: First, certain special “filenames” are recognized. If the filename used in the constructor or an `open` call is `cin>, cout>, cerr>,` or `clog>` (the angle brackets must be part of the string), then the corresponding standard stream of the same name is used for input (`pt_ifstream`) or output (`pt_ofstream`). In addition, C standard I/O fans can specify `stdin>, stdout>,` or `stderr>` as well. Second, the Ptolemy `expandPathName` is applied to the filename before it is opened, permitting it to start with `~user` or `$VAR`. Finally, if a failure occurs when the file is opened, `Error::abortRun` is called with an appropriate error message, including the Unix error condition. Otherwise these classes are identical to the standard ifstream and ofstream classes and can be used as replacements.

11.5 XGraph, an interface to the xgraph program

The XGraph class provides an interface for the xgraph program for plotting data on an X window system display. The modified xgraph program provided with the Ptolemy distribution should be used, not the contributed version from the X11R5 tape. The constructor for XGraph does not completely initialize the object; initialization is completed by the initialize() method:

```
void initialize(Block* parent, int noGraphs,
     const char* options, const char* title,
     const char* saveFile = 0, int ignore = 0);
```

The *parent* argument is the name of a Block that is associated with the XGraph object; this Block is used in Error::abortRun messages to report errors. *noGraphs* specifies the number of data sets that the graph will contain. Each data set is a separate stream and is plotted in a different color (a different line style for B/W displays). *options* is a series of command line options that will be passed unmodified to the xgraph program. It is subject to expansion by the Unix shell. *title* is the title for the graph; it can contain special characters (it is *not* subjected to expansion by the Unix shell). *saveFile* is the name of a file to save the graph data into, in ASCII form. If it is not given, the data are not saved, and a faster binary format is used. *ignore* specifies the number of initial points to ignore from each data set.

```
void setIgnore(int n);
```

Reset the "ignore" parameter to *n.*

```
void addPoint(float y);
```

Add a single point to the first data set whose X value is automatically generated (0, 1, 2, 3... on successive calls) and whose Y value is *y.*

```
void addPoint(float x,float y);
```

Add the point (*x, y)* to the first data set.

```
void addPoint(int dataSet,float x,float y);
```

Add the point (*x, y)* to the data set indicated by *dataSet.* Data sets start with 1.

```
void newTrace(int dataSet = 1);
```

Start a new trace for the nth dataset. This means that there will be no connecting line between the last point plotted and the next point plotted.

```
void terminate();
```

This function flushes the data out to disk, closes the files, and invokes the xgraph program. If the destructor is called before terminate, it will close and delete the temporary files.

11.6 Histogram classes

The Histogram class accepts a stream of data and accumulates a histogram. The XHistogram class uses a Histogram to collect the data and an XGraph to display it.

11.6.1 Class Histogram

The `Histogram` class accumulates data in a histogram. Its constructor is as follows:

`Histogram(double` *`width`* `= 1.0, int` *`maxBins`* `= HISTO_MAX_BINS);`

The default maximum number of bins is 1000. The bin centers will be at integer multiples of the specified bin width. The total width of the histogram depends on the data; however, there will always be a bin that includes the first point.

`void add(double` *`x`*`);`

Add the point *`x`* to the histogram.

```
int numCounts() const;
double mean() const;
double variance() const;
```

Return the number of counts, the mean, and the variance of the data in the histogram.

`int getData(int` *`binno`*`, int&` *`count`*`, double&` *`binCenter`*`);`

Get counts and bin centers by bin number, where 0 indicates the smallest bin. Return `TRUE` if this is a valid bin. Thus the entire histogram data can be retrieved by stepping from 0 to the first failure.

11.6.2 Class XHistogram

An `XHistogram` object has a private `XGraph` member and a private `Histogram` member. The functions

```
int numCounts() const;
double mean() const;
double variance();
```

simply pass through to the `Histogram` object, and

`void addPoint(float` *`y`*`);`

adds a point to the histogram and does other bookkeeping. There are two remaining methods:

```
void initialize(Block* parent, double binWidth,
   const char* options, const char* title,
   const char* saveFile, int maxBins = HISTO_MAX_BINS
```

This method initializes the graph and histogram object. *`parent`* is the parent `Block`, used for error messages. *`binWidth`* and *`maxBins`* initialize the `Histogram` object. *`options`* is a string that is included in the command line to the `xgraph` program; other options, including `-bar -nl -brw` value*,* are passed as well. *`title`* is the graph title, and *`saveFile,`* if non-null, gives a file in which the histogram data is saved (this data is the histogram counts, not the data that was input with `addPoint).`

`void terminate();`

This method completes the histogram, flushes out the temporary files, and executes `xgraph`.