Reference manual
to the GNU C++ Iostream Facility

(Very much a work in progress)

Per Bothner (bothner@cygnus.com)

1 Introduction

The `iostream` library was written by Per Bothner.

Various people have found bugs or come with suggestions. Hongjiu Lu has worked hard to use the library as the default stdio implementation for Linux, and has provided much stress-testing of the library.

Some code was derived from parts of BSD 4.4, which is copyright University of California at Berkeley.

2 Using the iostream layer

2.1 C-style formatting for streams

These methods all return `*this`.

`ostream& ostream::vform(const char *format, ...)` [Method]
> Similar to `fprintf(file, format, ...)`. The *format* is a `printf`-style format control string, which is used to format the (variable number of) arguments, printing the result on the `this` stream.

`ostream& ostream::vform(const char *format, va_list args)` [Method]
> Similar to `vfprintf(file, format, args)`. The *format* is a `printf`-style format control string, which is used to format the argument list *args*, printing the result on the `this` stream.

`istream& istream::scan(const char *format, ...)` [Method]
> Similar to `fscanf(file, format, ...)`. The *format* is a `scanf`-style format control string, which is used to read the (variable number of) arguments from the `this` stream.

`istream& istream::vscan(const char *format, va_list args)` [Method]
> Like `istream::scan`, but takes a single `va_list` argument.

3 Using the streambuf layer

3.1 C-style formatting for streambufs

The GNU streambuf class supports `printf`-like formatting.

int streambuf::vform(const char *_format_, ...)   [Method]
  Similar to `fprintf(`_file_`, `_format_`, ...)`. The _format_ is a `printf`-style format control string, which is used to format the (variable number of) arguments, printing the result on the `this` streambuf. The result is the number of characters printed.

int streambuf::vform(const char *_format_, va_list _args_)   [Method]
  Similar to `vfprintf(`_file_`, `_format_`, `_args_`)`. The _format_ is a `printf`-style format control string, which is used to format the argument list _args_, printing the result on the `this` streambuf. The result is the number of characters printed.

int streambuf::scan(const char *_format_, ...)   [Method]
  Similar to `fscanf(`_file_`, `_format_`, ...)`. The _format_ is a `scanf`-style format control string, which is used to read the (variable number of) arguments from the `this` streambuf. The result is the number of items assigned, or `EOF` in case of input failure before any conversion.

int streambuf::vscan(const char *_format_, va_list _args_)   [Method]
  Like `streambuf::scan`, but takes a single `va_list` argument.

3.2 stdiobuf

A _stdiobuf_ is a `streambuf` object that points to a `FILE` object (as defined by `stdio.h`). All `streambuf` operations on the `stdiobuf` are forwarded to the `FILE`. Thus the `stdiobuf` object provides a wrapper around a `FILE`, allowing use of `streambuf` operations on a `FILE`. This can be useful when mixing C code with C++ code.

The pre-defined streams `cin`, `cout`, and `cerr` are normally implemented as `stdiobuf`s that point to respectively `stdin`, `stdout`, and `stderr`. This is convenient, but it does cost some extra overhead. (If you have sets things up so that you use the implementation of stdio provided with this library, then `cin`, `cout`, and `cerr` will be set up to ot use `stdiobuf`s, since you get their benefits for free.)

Note that if you use `setbuf` to give a buffer to a `stdiobuf`, that buffer will provide intermediate buffering in addition that whatever is done by the `FILE`.

3.3 indirectbuf

An _indirectbuf_ is one that forwards all of its I/O requests to another streambuf. All get-related requests are sent to get_stream(). All put-related requests are sent to put_stream().

An indirectbuf can be used to implement Common Lisp synonym-streams and two-way-streams:

```
class synonymbuf : public indirectbuf {
    Symbol *sym;
```

```
        synonymbuf(Symbol *s) { sym = s; }
        virtual streambuf *lookup_stream(int mode) {
            return coerce_to_streambuf(lookup_value(sym)); }
    };
```

3.4 Backing up

The GNU iostream library allows you to ask streambuf to remember the current position, and then later after you've read further be able to go back to it. Your're guaranteed to be able to backup arbitrary amounts, even on unbuffered files or multiple buffers worth, as long as you tell the library advance. This unbounded backup is very useful for scanning and parsing applications. This example shows a typical scenario:

```
// Read either "dog", "hound", or "hounddog".
// If "dog" is found, return 1.
// If "hound" is found, return 2.
// If "hounddog" is found, return 3.
// If non of these are found, return -1.
int my_scan(streambuf* sb)
{
    streammarker fence(sb);
    char buffer[20];
    // Try reading "hounddog":
    if (sb->sgetn(buffer, 8) == 8 && strncmp(buffer, "hounddog", 8) == 0)
      return 3;
    // No, no "hounddog":  Backup to 'fence' ...
    sb->seekmark(fence); //
    // ... and try reading "dog":
    if (sb->sgetn(buffer, 3) == 3 && strncmp(buffer, "dog", 3) == 0)
      return 1;
    // No, no "dog" either:  Backup to 'fence' ...
    sb->seekmark(fence); //
    // ... and try reading "hound":
    if (sb->sgetn(buffer, 5) == 5 && strncmp(buffer, "hound", 5) == 0)
      return 2;
    // No, no "hound" either:  Backup to 'fence' and signal failure.
    sb->seekmark(fence); // Backup to 'fence'..
    return -1;
}
```

streammarker::streammarker (streambuf* *sbuf***)**   [Constructor]
    Create a **streammarker** associated with *sbuf* that remembers the current postion of the get pointer.

int streammarker::delta (streammarker& *mark2***)**   [Method]
    Return the difference between the get positions corresponding to ***this** and *mark2* (which must point into the same **streambuffer** as **this**).

`int streammarker::delta ()`   [Method]
      Return the position relative to the streambuffer's current get position.

`int streambuffer::seekmark (streammarker& `*mark*`)`   [Method]
      Move the get pointer to where it (logicly) was when *mark* was constructed.

4 stdio: C input/output

Iostreams is distributed with a complete implementation of the ANSI C stdio facility. It is implemented using streambufs.

The stdio package is intended as a replacement for the whatever stdio is in your C library. It can co-exist with C libraries that have alternate implementations of stdio, but there may be some problems. Since stdio works best when you build libc to contain it, and that may be inconvenient, it is not installed by default.

Extensions beyond ANSI:

- A stdio FILE is identical to a streambuf. Hence there is no need to worry about synchronizing C and C++ input/output - they are by definition always synchronized.

- If you create a new streambuf sub-class (in C++), you can use it as a FILE from C. Thus the system is extensible using the standard streambuf protocol.

- You can arbitrarily mix reading and writing, without having to seek in between.

- Unbounded ungetc() buffer.

5 Streambuf internals

5.1 Buffer management

5.1.1 Areas

Streambuf buffer management is fairly sophisticated (this is a nice way to say "complicated"). The standard protocol has the following "areas":

- The *put area* contains characters waiting for output.
- The *get area* contains characters available for reading.
- The *reserve area* is available to virtual methods. Usually, the get and/or put areas are part of the reserve area.

  The GNU streambuf design extends this by supporting two get areas:

- The *main get area* contains characters that have been read in from the character source, but not yet read by the application.
- The *backup area* contains previously read data that is being saved because of a user request, or that have been "unread" (putback).

  The backup and the main get area are logically contiguous: That is, the first character of the main get area follows the last character of the backup area.

  The *current get area* is whichever one of the backup or main get areas that is currently being read from. The other of the two is the *non-current get area*.

5.1.2 Pointers

The following `char*` pointers define the various areas. (Note that if a pointer points to the 'end' of an area, it means that it points to the character after the area.)

`char* streambuffer::base ()`   [Method]
>   The start of the reserve area.

`char* streambuffer::ebuf ()`   [Method]
>   The end of the reserve area.

`char* streambuffer::pbase ()`   [Method]
>   The start of the put area.

`char* streambuffer::pptr ()`   [Method]
>   The current put position. If `pptr() < epptr()`, then the next write will overwrite `*pptr()`, and increment `pptr()`.

`char* streambuffer::epptr ()`   [Method]
>   The end of the put area.

`char* streambuffer::eback ()`   [Method]
>   The start of the current get area.

`char* streambuffer::gptr ()`   [Method]
> The current get position. If `gptr() < egptr()`, then the next read will read `*gptr()`, and increment `gptr()`.

`char* streambuffer::egptr ()`   [Method]
> The end of the current get area.

`char* streambuffer::Gbase ()`   [Method]
> The start of the main get area.

`char* streambuffer::eGptr ()`   [Method]
> The end of the main get area.

`char* streambuffer::Bbase ()`   [Method]
> The start of the backup area.

`char* streambuffer::Bptr ()`   [Method]
> The start of the used part of the backup area. The area (`Bptr() .. eBptr()`) contains data that has been pushed back, while (`Bbase() .. eBptr()`) contains unused space available for future putbacks.

`char* streambuffer::eBptr ()`   [Method]
> The end of the backup area.

`char* streambuffer::Nbase ()`   [Method]
> The start of the non-current get area (either `main_gbase` or `backup_gbase`).

`char* streambuffer::eNptr ()`   [Method]
> The end of the non-current get area.

5.2 Filebuf internals

The `filebuf` is used a lot, so it is importamt that it be efficient. It is also supports rather complex semantics. so let us examine its implementation.

5.2.1 Tied read and write pointers

The streambuf model allows completely independent read and write pointers. However, a `filebuf` has only a single logical pointer used for both reads and writes. Since the `streambuf` protocol uses `gptr()` for reading and `pptr()` for writing, we map the logical file pointer into either `gptr()` or `pptr()` at different times.

- Reading is allowed when `gptr() < egptr()`, which we call get mode.
- Writing is allowed when `pptr() < epptr()`, which we call put mode.

A `filebuf` cannot be in get mode and put mode at the same time.

We have upto two buffers:

- The backup area, defined by `Bbase()`, `Bptr()`, and `eBptr()`. This can be empty.
- The reserve area, which also contains the main get area. For an unbuffered file, the (`shortbuf()..shortbuf()+1`) is used, where `shortbuf()` points to a 1-byte buffer that is part of the `filebuf`.

The file system's idea of the current position is `eGptr()`.

Character that have been written into a buffer but not yet written out (flushed) to the file systems are those between `pbase()` and `pptr()`.

The end of the valid data bytes is: `pptr() > eGptr() && pptr() < ebuf() ? pptr() : eGptr()`.

If the `filebuf` is unbuffered or line buffered, the `eptr()` is `pbase()`. This forces a call to `overflow()` on each put of a character. The logical `epptr()` is `epptr() ? ebuf() : NULL`. (If the buffer is read-only, set `pbase()`, `pptr()`, and `epptr()` to `NULL`. NOT!)

Appendix A Function and Variable Index,Concept Index,,Top

Appendix B Concept Index,,Function and Variable Index,Top

Table of Contents