C++ Socket Classes

Version: 07Nov93 1.5

Gnanasekaran Swaminathan

This is Version: 07Nov93 1.5 of the C++ family of socket classes.

Socket++ Library Copyright Notice

Acknowledgments

Gordon Joly <G.Joly@cs.ucl.ac.uk> for reporting bugs in pipestream class implementation and providing an ftp site for the socket++ library at cs.ucl.ac.uk:~ftp/coside/gnu/socket++-1.x.tar.gz He also knows how to make the socket++ library a shared library.

Jim Anderson for reporting a bug in sockinet.C

Carl Gay <cgay@skinner.cs.uoregon.edu> for reporting a bug and a fix in sockinet.C

Oliver Imbusch <flabes@parystec.de> for reporting a bug in Makefile.in and suggesting several enhancements for sockbuf class.

Per Bothner <bothner@cygnus.com> for configure, config.sub, config.shared and move-if-change files that are used to generate Makefile. These files are taken from his libg++-2.4 and hence, these files are governed by the Copyright Notice found in the file LICENCE in libg++.

1 Overview of Socket++ Library

Socket++ library defines a family of C++ classes that can be used more effectively than directly calling the underlying low-level system functions. One distinct advantage of the socket++ is that it has the same interface as that of the iostream so that the users can perform type-safe input output. See ⟨undefined⟩ [IOStream], page ⟨undefined⟩, for more information on iostream library.

`streambuf` counterpart of the socket++ is `sockbuf`. `sockbuf` is an endpoint for communication with yet another `sockbuf` or simply a `socket` descriptor. `sockbuf` has also methods that act as interfaces for most of the commonly used system calls that involve sockets. See Chapter 2 [sockbuf Class], page 4, for more information on the socket buffer class.

For each communication domain, we derive a new class from `sockbuf` that has some additional methods that are specific to that domain. At present, only *unix* and *inet* domains are supported. `sockunixbuf` class and `sockinetbuf` class define the *unix* and *inet* domain of sockets respectively. See Chapter 6 [sockunixbuf Class], page 21, for *unix* sockets and See Chapter 4 [sockinetbuf Class], page 14, for *inet* sockets.

We also have domain specific socket address classes that are derived from a common base class called `sockAddr`. `sockunixaddr` class is used for *unix* domain addresses and `sockinetaddr` class is used for *inet* domain addresses. For more information on address classes see Chapter 3 [sockAddr Class], page 13, Chapter 7 [sockunixaddr Class], page 25, and Chapter 5 [sockinetaddr Class], page 19.

> *Note*: `sockAddr` is not spelled `sockaddr` in order to prevent name clash with the `struct sockaddr` declared in '<sys/socket.h>'.

We noted earlier that socket++ provides the same interface as the iostream library. For example, in the internet domain, we have `isockinet`, `osockinet`, and `iosockinet` classes that are counterparts to `istream`, `ostream`, and `iostream` classes of `libg++ iostream` library. For more details on `iosockstream` classes see See Chapter 8 [sockstream Classes], page 26.

The services of `pipe()`, `socketpair()`, and `popen()` are provided by the `pipestream` class. See Chapter 9 [pipestream Classes], page 33.

2 sockbuf Class

`sockbuf` class is derived from `streambuf` class of the libg++ iostream library. You can simultaneously read and write into a `sockbuf` just like you can listen and talk through a telephone. To accomplish the above goal, we maintain two independent buffers for reading and writing.

2.1 Constructors

`sockbuf` constructors sets up an endpoint for communication. A `sockbuf` object so created can be read from and written to in linebuffered mode. To change mode, refer to `streambuf` class in See ⟨undefined⟩ [IOStream], page ⟨undefined⟩.

`sockbuf` objects are created as follows where

- `s` and `so` are `sockbuf` objects
- `sd` is an integer which is a socket descriptor
- `af` and `proto` are integers which denote domain number and protocol number respectively
- `ty` is a `sockbuf::type` and must be one of `sockbuf::sock_stream`, `sockbuf::sock_dgram`, `sockbuf::sock_raw`, `sockbuf::sock_rdm`, and `sockbuf::sock_seqpacket`

`sockbuf s(sd);`
`sockbuf s;`
: Set socket descriptor of `s` to `sd` (defaults to -1). `sockbuf` destructor will close `sd`.

`sockbuf s(af, ty, proto);`
: Set socket descriptor of `s` to `::socket(af, int(ty), proto);`

`sockbuf so(s);`
: Set socket descriptor of `so` to the socket descriptor of `s`.

`s.open(ty, proto)`
: does nothing and returns simply `0`, the null pointer to `sockbuf`.

`s.is_open()`
: returns a non-zero number if the socket descriptor is open else return 0.

`s = so;`
: return a reference `s` after assigning `s` with `so`.

2.2 Destructor

sockbuf::~sockbuf() flushes output and closes its socket if no other sockbuf is referencing it and _S_DELETE_DONT_CLOSE flag bit of streambuf base class is not set. It also deletes its read and write buffers.

In what follows,

- s is a sockbuf object
- how is of type sockbuf::shuthow and must be one of sockbuf::shut_read, sockbuf::shut_write, and sockbuf::shut_readwrite

sockbuf::~sockbuf()
: flushes output and closes its socket if no other sockbuf object is referencing it before deleting its read and write buffers. If the _S_DELETE_DONT_CLOSE bit of streambuf base class is set, then the socket is not closed.

s.close()
: closes the socket even if it is referenced by other sockbuf objects and _S_DELETE_DONT_CLOSE flag is set.

s.shutdown(how)
: shuts down read if how is sockbuf::shut_read, shuts down write if how is sockbuf::shut_write, and shuts down both read and write if how is sockbuf::shut_readwrite.

2.3 Reading and Writing

sockbuf class offers several ways to read and write and tailors the behavior of several virtual functions of streambuf for socket communication.

In case of error, sockbuf::error(const char*) is called.

In what follows,

- s is a sockbuf object
- buf is buffer of type char*
- bufsz is an integer and is less than sizeof(buf)
- msgf is an integer and denotes the message flag

- sa is of type sockAddr
- msgh is a pointer to struct msghdr
- wp is an integer and denotes time in seconds
- c is a char

`s.is_open()`
: returns a non-zero number if the socket descriptor is open else return 0.

`s.is_eof()`
: returns a non-zero number if the socket has seen EOF while reading else return 0.

`s.write(buf, bufsz)`
: returns an int which must be equal to bufsz if bufsz chars in the buf are written successfully. It returns 0 if there is nothing to write or if, in case of timeouts, the socket is not ready for write Section 2.6 [Timeouts], page 11.

`s.send(buf, bufsz, msgf)`
: same as sockbuf::write described above but allows the user to control the transmission of messages using the message flag msgf. If msgf is sockbuf::msg_oob and the socket type of s is sockbuf::sock_stream, s sends the message in *out-of-band* mode. If msgf is sockbuf::msg_dontroute, s sends the outgoing packets without routing. If msgf is 0, which is the default case, sockbuf::send behaves exactly like sockbuf::write.

`s.sendto(sa, buf, bufsz, msgf)`
: same as sockbuf::send but works on unconnected sockets. sa specifies the *to* address for the message.

`s.sendmsg(msgh, msgf)`
: same as sockbuf::send but sends a struct msghdr object instead.

`s.sys_write(buf, bufsz)`
: calls sockbuf::write and returns the result. Unlike sockbuf::write sockbuf::sys_write is declared as a virtual function.

`s.read(buf, bufsz)`
: returns an int which is the number of chars read into the buf. In case of EOF, return EOF. Here, bufsz indicates the size of the buf. In case of timeouts, return 0 Section 2.6 [Timeouts], page 11.

`s.recv(buf, bufsz, msgf)`
: same as sockbuf::read described above but allows the user to receive *out-of-band* data if msgf is sockbuf::msg_oob or to preview the data waiting to be read if msgf is sockbuf::msg_peek. If msgf is 0, which is the default case, sockbuf::recv behaves exactly like sockbuf::read.

`s.recvfrom(sa, buf, bufsz, msgf)`

same as `sockbuf::recv` but works on unconnected sockets. `sa` specifies the *from* address for the message.

`s.recvmsg(msgh, msgf)`

same as `sockbuf::recv` but reads a `struct msghdr` object instead.

`s.sys_read(buf, bufsz)`

calls `sockbuf::read` and returns the result. Unlike `sockbuf::read` `sockbuf::sys_read` is declared as a virtual function.

`s.is_readready(wp_sec, wp_usec)`

returns a non-zero int if `s` has data waiting to be read from the communication channel. If `wp_sec >= 0`, it waits for `wp_sec 10^6 + wp_usec` microseconds before returning 0 in case there are no data waiting to be read. If `wp_sec < 0`, then it waits until a datum arrives at the communication channel. `wp_usec` defaults to 0.

> *Please Note*: The data waiting in `sockbuf`'s own buffer is different from the data waiting in the communication channel.

`s.is_writeready(wp_sec, wp_usec)`

returns a non-zero int if data can be written onto the communication channel of `s`. If `wp_sec >= 0`, it waits for `wp_sec 10^6 + wp_usec` microseconds before returning 0 in case no data can be written. If `wp_sec < 0`, then it waits until the communication channel is ready to accept data. `wp_usec` defaults to 0.

> *Please Note*: The buffer of the `sockbuf` class is different from the buffer of the communication channel buffer.

`s.is_exceptionpending(wp_sec, wp_usec)`

returns non-zero int if `s` has any exception events pending. If `wp_sec >= 0`, it waits for `wp_sec 10^6 + wp_usec` microseconds before returning 0 in case `s` does not have any exception events pending. If `wp_sec < 0`, then it waits until an expception event occurs. `wp_usec` defaults to 0.

> *Please Note*: The exceptions that `sockbuf::is_exceptionpending` is looking for are different from the C++ exceptions.

`s.flush_output()`

flushes the output buffer and returns the number of chars flushed. In case of error, return EOF. `sockbuf::flush_output` is a protected member function and it is not available for general public.

`s.doallocate()`

allocates free store for read and write buffers of `s` and returns 1 if allocation is done and returns 0 if there is no need. `sockbuf::doallocate` is a protected virtual member function and it is not available for general public.

`s.underflow()`

returns the unread char in the buffer as an unsigned char if there is any. Else returns EOF if `s` cannot allocate space for the buffers, cannot read or peer is closed. `sockbuf::underflow` is a protected virtual member function and it is not available for general public.

`s.overflow(c)`

if `c=EOF`, call and return the result of `flush_output()`, else if `c='\n'` and `s` is linebuffered, call `flush_output()` and return `c` unless `flush_output()` returns EOF, in which case return EOF. In any other case, insert char `c` into the buffer and return `c` as an unsigned char. `sockbuf::overflow` is a protected member virtual function and it is not available for general public.

`s.sync()` calls `flush_output()` and returns the result. Useful if the user needs to flush the output without writing newline char into the write buffer.

`s.xsputn(buf, bufsz)`

write `bufsz` chars into the buffer and returns the number of chars successfully written. Output is flushed if any char in `buf[0..bufsz-1]` is `'\n'`.

`s.recvtimeout(wp)`

sets the recv timeout to `wp` seconds. If `wp` is -1, it is a block and if `wp` is 0, it is a poll.

It affects all read functions. If the socket is not read ready within `wp` seconds, the read call will return 0. It also affects `sockbuf::underflow`. `sockbuf::underflow` will not set the `_S_EOF_SEEN` flag if it is returning EOF because of timeout.

`sockbuf::recvtimeout` returns the old recv timeout value.

`s.sendtimeout(wp)`

sets the send timeout to `wp` seconds. If `wp` is -1, it is a block and if `wp` is 0, it is a poll.

It affects all write functions. If the socket is not write ready within `wp` seconds, the write call will return 0.

`sockbuf::sendtimeout` returns the old send timeout value.

2.4 Establishing connections

A name must be bound to a `sockbuf` if processes want to refer to it and use it for communication. Names must be unique. A *unix* name is a 3-tuple, <*protocol, local path, peer path*>. An *inet* name is a 5-tuple, <*protocol, local addr, local port, peer addr, peer port*>. `sockbuf::bind` is used to specify the local half of the name—<*local path*> for *unix* and <*local addr, local port*> for *inet*. `sockbuf::connect` and `sockbuf::accept` are used to specify the peer half of the name—<*peer path*> for *unix* and <*peer addr, peer port*> for *inet*.

In what follows,

- s and so are sockbuf objects
- sa is a sockAddr object
- nc is an integer denoting the number of connections to allow

`s.bind(sa)`
: binds sockAddr sa as the local half of the name for s.

`s.connect(sa)`
: sockbuf::connect uses sa to provide the peer half of the name for s and to establish the connection itself. sockbuf::connect also provides the local half of the name automatically and hence, the user should not use sockbuf::bind to bind any local half of the name.

`s.listen(nc)`
: makes s ready to accept connections. nc specifies the maximum number of outstanding connections that may be queued and must be at least 1 and less than or equal to sockbuf::somaxconn which is usually 5 on most systems.

`sockbuf so = s.accept(sa)`
`sockbuf so = s.accept()`
: accepts connections and returns the peer address in sa. s must be a listening sockbuf. See sockbuf::listen above.

2.5 Getting and Setting Socket Options

Socket options are used to control a socket communication. New options can be set and old value of the options can be retrived at the protocol level or at the socket level by using setopt and getopt member functions. In addition, you can also use special member functions to get and set specific options.

In what follows,

- s is a sockbuf object
- opval is an integer and denotes the option value
- op is of type sockbuf::option and must be one of
  - sockbuf::so_error used to retrieve and clear error status
  - sockbuf::so_type used to retrieve type of the socket

- sockbuf::so_debug is used to specify recording of debugging information
- sockbuf::so_reuseaddr is used to specify the reuse of local address
- sockbuf::so_keepalive is used to specify whether to keep connections alive or not
- sockbuf::so_dontroute is used to specify whether to route messages or not
- sockbuf::so_broadcast is used to specify whether to broadcast sockbuf::sock_dgram messages or not
- sockbuf::so_oobinline is used to specify whether to inline *out-of-band* data or not.
- sockbuf::so_linger is used to specify for how long to linger before shutting down
- sockbuf::so_sndbuf is used to retrieve and to set the size of the send buffer (communication channel buffer not sockbuf's internal buffer)
- sockbuf::so_rcvbuf is used to retrieve and to set the size of the recv buffer (communication channel buffer not sockbuf's internal buffer)

s.getopt(op, &opval, sizeof(opval), oplevel)

gets the option value of the sockbuf::option op at the option level oplevel in opval. It returns the actual size of the buffer opval used. The default value of the oplevel is sockbuf::sol_socket.

s.setopt(op, &opval, sizeof(opval), oplevel)

sets the option value of the sockbuf::option op at the option level oplevel to opval. The default value of the oplevel is sockbuf::sol_socket.

s.gettype()

gets the socket type of s. The return type is sockbuf::type.

s.clearerror()

gets and clears the error status of the socket.

s.debug(opval)

if opval is not -1, set the sockbuf::so_debug option value to opval. In any case, return the old option value of sockbuf::so_debug option. The default value of opval is -1.

s.reuseaddr(opval)

if opval is not -1, set the sockbuf::so_reuseaddr option value to opval. In any case, return the old option value of sockbuf::so_reuseaddr option. The default value of opval is -1.

s.dontroute(opval)

if opval is not -1, set the sockbuf::so_dontroute option value to opval. In any case, return the old option value of sockbuf::so_dontroute option. The default value of opval is -1.

`s.oobinline(opval)`

: if `opval` is not -1, set the `sockbuf::so_oobinline` option value to `opval`. In any case, return the old option value of `sockbuf::so_oobinline` option. The default value of `opval` is -1.

`s.broadcast(opval)`

: if `opval` is not -1, set the `sockbuf::so_broadcast` option value to `opval`. In any case, return the old option value of `sockbuf::so_broadcast` option. The default value of `opval` is -1.

`s.keepalive(opval)`

: if `opval` is not -1, set the `sockbuf::so_keepalive` option value to `opval`. In any case, return the old option value of `sockbuf::so_keepalive` option. The default value of `opval` is -1.

`s.sendbufsz(opval)`

: if `opval` is not -1, set the new send buffer size to `opval`. In any case, return the old buffer size of the send buffer. The default value of `opval` is -1.

`s.recvbufsz(opval)`

: if `opval` is not -1, set the new recv buffer size to `opval`. In any case, return the old buffer size of the recv buffer. The default value of `opval` is -1.

`s.linger(tim)`

: if `tim` is positive, set the linger time to tim seconds. If `tim` is 0, set the linger off. In any case, return the old linger time if it was set earlier. Otherwise return -1. The default value of `tim` is -1.

2.6 Time Outs While Reading and Writing

Time outs are very useful in handling data of unknown sizes and formats while reading and writing. For example, how does one communicate with a socket that sends chunks of data of unknown size and format? If only `sockbuf::read` is used without time out, it will block indefinitely. In such cases, time out facility is the only answer.

The following idiom is recommended.

```
int old_tmo = s.recvtimeout (2) // set time out (2 seconds here)
for (;;) { // read or write
    char buf[256];
    int rval = s.read (buf, 256);
    if (rval == 0 || rval == EOF) break;
```

```
        // process buf
    }
    s.recvtimeout (old_tmo); // reset time out
```

In what follows,

- `s` is a `sockbuf` object
- `wp` is waiting period in seconds

`s.recvtimeout(wp)`

sets the recv timeout to `wp` seconds. If `wp` is -1, it is a block and if `wp` is 0, it is a poll.

It affects all read functions. If the socket is not read ready within `wp` seconds, the read call will return 0. It also affects `sockbuf::underflow`. `sockbuf::underflow` will not set the `_S_EOF_SEEN` flag if it is returning EOF because of timeout.

`sockbuf::recvtimeout` returns the old recv timeout value.

`s.sendtimeout(wp)`

sets the send timeout to `wp` seconds. If `wp` is -1, it is a block and if `wp` is 0, it is a poll.

It affects all write functions. If the socket is not write ready within `wp` seconds, the write call will return 0.

`sockbuf::sendtimeout` returns the old send timeout value.

3 sockAddr Class

Class `sockAddr` is an abstract base class for all socket address classes. That is, domain specific socket address classes are all derived from `sockAddr` class.

> *Note*: `sockAddr` is not spelled `sockaddr` in order to prevent name clash with `struct sockaddr` declared in '`<sys/socket.h>`'.

Non-abstract derived classes must have definitions for the following functions.

`sockAddr::operator void* ()`
: should simply return `this`.

`sockAddr::size()`
: should return `sizeof(*this)`. The return type is `int`.

`sockAddr::family()`
: should return address family (domain name) of the socket address. The return type is `int`

4 sockinetbuf Class

`sockinetbuf` class is derived from `sockbuf` class and inherits most of the public functions of `sockbuf`. See Chapter 2 [sockbuf Class], page 4, for more information on `sockbuf`. In addition, it provides methods for getting `sockinetaddr` of local and peer connections. See Chapter 5 [sockinetaddr Class], page 19, for more information on `sockinetaddr`.

4.1 Methods

In what follows,

- `ty` denotes the type of the socket connection and is of type `sockbuf::type`
- `proto` denotes the protocol and is of type int
- `si` is a `sockbuf` object and is in *inet* domain

`sockinetbuf ins(ty, proto)`
: Constructs a `sockinetbuf` object `ins` whose socket communication type is `ty` and protocol is `proto`. `proto` defaults to 0.

`sockinetbuf ins(si)`
: Constructs a `sockinetbuf` object `ins` which uses the same socket as `si` uses.

`ins = si`
: performs the same function as `sockbuf::operator=`. See Chapter 2 [sockbuf Class], page 4, for more details.

`ins.open(ty, proto)`
: create a new `sockinetbuf` whose type and protocol are `ty` and `proto` respectively and assign it to `ins`.

`sockinetaddr sina = ins.localaddr()`
: returns the local *inet* address of the `sockinetbuf` object `ins`. The call will make sense only after a call to either `sockbuf::bind` or `sockbuf::connect`.

`sockinetaddr sina = ins.peeraddr()`
: returns the peer *inet* address of the `sockinetbuf` object `ins`. The call will make sense only after a call to `sockbuf::connect`.

`const char* hn = ins.localhost()`
: returns the local *inet* thostname of the `sockinetbuf` object `ins`. The call will make sense only after a call to either `sockbuf::bind` or `sockbuf::connect`.

`const char* hn = ins.peerhost()`

returns the peer *inet* thostname of the `sockinetbuf` object `ins`. The call will make sense only after a call to `sockbuf::connect`.

`int pn = ins.localport()`

returns the local *inet* port number of the `sockinetbuf` object `ins` in host byte order. The call will make sense only after a call to either `sockbuf::bind` or `sockbuf::connect`.

`int pn = ins.peerport()`

returns the peer *inet* port number of the `sockinetbuf` object `ins` in local host byte order. The call will make sense only after a call to `sockbuf::connect`.

4.2 *inet* Datagram Sockets

The following two programs illustrates how to use `sockinetbuf` class for datagram connection in *inet* domain. `tdinread.cc` also shows how to use `isockstream` class and `tdinwrite.cc` shows how to use `osockstream` class.

tdinread.cc

```
// reads data sent by tdinwrite.cc
#include <sockinet.h>

main(int ac, char** av)
{
        sockinetbuf  so (sockbuf::sock_dgram);
        sockinetaddr sina;

        so.bind(sina);

        cout << "localhost = " << so.localhost() << endl
             << "localport = " << so.localport() << endl;

        isockstream  is(so);
        char         buf[256];
        int          n;

        is >> n;
        cout << av[0] << ": ";
        while(n--) {
                is >> buf;
                cout << buf << ' ';
```

```
        }
        cout << endl;
}
```

tdinwrite.cc

```
// sends data to tdinread.cc
#include <sockinetbuf.h>

extern "C" int        atoi(const char*);

main(int ac, char** av)
{
        if (ac < 3) {
                cerr << "USAGE: " << av[0] << " thostname port-number "
                     << "data ... " << endl;
                return 1;
        }

        sockinetbuf     so(sockbuf::sock_dgram);
        sockinetaddr    sina(av[1], atoi(av[2]));

        so.connect(sina);

        cout << "local: " << so.localport() << ' '
                          << so.localhost() << endl
             << "peer:  " << so.peerport() << ' '
                          << so.peerhost() << endl;
        osockstream os(so);
        os << ac-3; av += 3;
        while(*av) os << *av++ << ' ';
        os << endl;
}
```

4.3 *inet* Stream Sockets

The following two programs illustrates the use of `sockinetbuf` class for stream connection in *inet* domain. It also shows how to use `iosockstream` class.

tsinread.cc

```
// receives strings from tsinwrite.cc and sends the strlen
// of each string back to tsinwrite.cc
#include        <sockinet.h>

main()
{
        sockinetaddr    addr;
        sockinetbuf     si(sockbuf::sock_stream);
        si.bind(addr);

        cout << si.localhost() << ' ' << si.localport() << endl;
        si.listen();
        sockinetbuf sin = si.accept();

        char          buf[1024];
        iosockstream  s(sin);

        while (s >> buf) {
                cout << buf << ' ';
                s << ::strlen(buf) << endl;
        }
        cout << endl;
}
```

tsinwrite.cc

```
// sends strings to tsinread.cc and gets back their length
// usage: tsinread hostname portno
//         see the output of tsinread for what hostname and portno to use

#include        <sockinet.h>

extern "C" int atoi(const char*);

main(int ac, char** av)
{
        sockinetaddr    addr(av[1], atoi(av[2]) );
        sockinetbuf     si(sockbuf::sock_stream);

        si.connect(addr);
        iosockstream s(si);
        s << "Hello! This is a test\n";

        // terminate the while loop in tsinread.cc
        si.shutdown(sockbuf::shut_write);

        int len;
        while (s >> len) cout << len << ' ';
```

```
        cout << endl;
}
```

5 sockinetaddr Class

Class `sockinetaddr` is derived from `sockAddr` declared in `<sockstream.h>` and from `sockaddr_in` declared in `<netinet/in.h>`. Always use a `sockinetaddr` object for an address with *inet* domain of sockets. See Section 2.4 [Connection Establishment], page 8.

In what follows,

- `adr` denotes an *inet* address in host byte order and is of type unsigned long
- `serv` denotes a service like "nntp" and is of type char*
- `proto` denotes a protocol like "tcp" and is of type char*
- `thostname` is of type char* and denotes the name of a host like `"kelvin.acc.virginia.edu"` or `"128.143.24.31"`.
- `portno` denotes a port in host byte order and is of type int

`sockinetaddr sina`

: Constructs a `sockinetaddr` object `sina` with default address *INADDR_ANY* and default port number 0.

`sockinetaddr sina(adr, portno)`

: Constructs a `sockinetaddr` object `sina` setting inet address to `adr` and the port number to `portno`. `portno` defaults to 0.

`sockinetaddr sina(adr, serv, proto)`

: Constructs a `sockinetaddr` object `sina` setting inet address to `adr` and the port number corresponding to the service `serv` and the protocol `proto`. The protocol defaults to "tcp".

`sockinetaddr sina(thostname, portno)`

: Constructs a `sockinetaddr` object `sina` setting inet address to the address of `thostname` and the port number to `portno`. `portno` defaults to 0.

`sockinetaddr sina(thostname, serv, proto)`

: Constructs a `sockinetaddr` object `sina` setting inet address to the address of `thostname` and the port number corresponding to the service `serv` and the protocol `proto`. The protocol defaults to "tcp".

`void* a = sina`

: returns the address of the `sockaddr_in` part of `sockinetaddr` object `sina` as void*.

`int sz = sina.size()`

: returns the sizeof `sockaddr_in` part of `sockinetaddr` object `sina`.

`int af = sina.family()`

returns `sockinetbuf::af_inet` if all is well.

`int pn = sina.getport()`

returns the port number of the `sockinetaddr` object `sina` in host byte order.

`const char* hn = getthostname()`

returns the host name of the `sockinetaddr` object `sina`.

6 sockunixbuf Class

`sockunixbuf` class is derived from `sockbuf` class declared in `<sockstream.h>` and hence, inherits most of the public member functions of `sockbuf`. See Chapter 2 [sockbuf Class], page 4, for more information on `sockbuf`.

6.1 Methods

In what follows,

- `ty` denotes the socket type and is of type `sockbuf::type`
- `proto` denotes the protocol number and is of type int
- `su` is a `sockbuf` and must be in *unix* domain

`sockunixbuf uns(ty, proto)`
: Constructs a `sockunixbuf` object `uns` with `ty` as its type and `proto` as its protocol number. `proto` defaults to 0.

`sockunixbuf uns = su`
: Constructs a `sockunixbuf` object `uns` which uses the same socket as is used by `su`.

`uns = su`
: `sockunixbuf` object `uns` closes its current socket if no other `sockbuf` is referring to it and uses the socket that `sockbuf` object `su` is using.

`uns.open(ty, proto)`
: create a `sockunixbuf` object with `ty` as its type and `proto` as its protocol and assign the `sockunixbuf` object so created to `*this`. It returns `this`. `proto` defaults to 0.

6.2 *unix* Datagram Sockets

The following two programs illustrates how to use `sockunixbuf` class for datagram connection in *unix* domain. `tdunread.cc` also shows how to use `isockstream` class and `tdunwrite.cc` shows how to use `osockstream` class.

tdunread.cc

```
// reads data sent by tdunwrite.cc
#include <sockunix.h>

extern "C" {
        int  chmod(const char* path, int mode);
        void perror(const char*);
        int  unlink(const char* path);
}

main(int ac, char** av)
{
        if (ac != 2) {
                cerr << "USAGE: " << av[0] << " socket_path_name\n";
                return 1;
        }

        sockunixbuf  sunb(sockbuf::sock_dgram);
        sockunixaddr suna(av[1]);

        sunb.bind(suna);
        cout << "Socket name = " << av[1] << endl;

        if (chmod(av[1], 0777) == -1) {
                perror("chmod");
                return 1;
        }

        isockstream isun(sunb);

        char buf[1024];
        int i;
        isun >> i;
        cout << av[0] << ": " << i << " strings: ";
        while (i--) {
                isun >> buf;
                cout  << buf << ' ';
        }
        cout << endl;

        unlink(av[1]);
}
```

tdunwrite.cc

```
// sends data to tdunread.cc
#include <sockunix.h>

main(int ac, char** av)
```

```
{
        if (ac < 2) {
                cerr << "USAGE: " << av[0]
                     << " socket_path_name data...\n";
                return 1;
        }

        sockunixbuf  sunb(sockbuf::sock_dgram);
        sockunixaddr suna(av[1]);

        sunb.connect(suna);
        osockstream       osun(sunb);
        osun << ac << ' ';
        for (int i=0; av[i]; i++) osun << av[i] << ' ';
        osun << endl;

}
```

6.3 *unix* Stream Sockets

The following two programs illustrates how to use `sockunixbuf` class for stream connection in *unix* domain. It also shows how to use `iosockstream` class.

tsunread.cc

```
// exchanges char strings with tsunwrite.cc
#include <sockunix.h>

extern "C" {
        int  chmod(const char* path, int mode);
        void perror(const char*);
        int  unlink(const char* path);
}

main(int ac, char** av)
{
        if (ac != 2) {
                cerr << "USAGE: " << av[0] << " socket_path_name\n";
                return 1;
        }

        sockunixbuf  su(sockbuf::sock_stream);
        sockunixaddr suna(av[1]);
```

```
        su.bind(suna);
        cout << "Socket name = " << av[1] << endl;

        if (chmod(av[1], 0777) == -1) {
                perror("chmod");
                return 1;
        }

        su.listen(3);
        sockunixbuf so = su.accept();

        iosockstream ioput(so);
        char         buf[1024];

        ioput << av[0] << ' ' << av[1] << endl;
        while ( ioput >> buf ) cout << av[0] << ": " << buf << endl;
        unlink(av[1]);
}
```

tsunwrite.cc

```
// exchanges char strings with tsunread.cc
#include <sockunix.h>

main(int ac, char** av)
{
        if (ac < 2) {
                cerr << "USAGE: " << av[0]
                     << " socket_path_name data...\n";
                return 1;
        }

        sockunixbuf  su(sockbuf::sock_stream);
        sockunixaddr suna(av[1]);

        su.connect(suna);
        iosockstream oput(su);
        char buf[128];

        oput >> buf;
        cout << buf << ' ';
        oput >> buf;
        cout << buf << endl;

        while (*av) oput << *av++ << ' ';
        oput << endl;
}
```

7 sockunixaddr Class

Class `sockunixaddr` is derived from class `sockAddr` declared in `<sockstream.h>` and from struct `sockaddr_un` declared in `<sys/un.h>`. Always use `sockunixaddr` objects for addresses with *unix* domain of sockets. See Section 2.4 [Connection Establishment], page 8.

In what follows,

- `path` is the *unix* path name like "/tmp/unix_socket"

`sockunixaddr suna(path)`
: Constructs a `sockunixaddr` object `suna` with `path` as the *unix* path name.

`void* a = suna`
: returns the address of the `sockaddr_un` part of `sockunixaddr` object `suna` as void*.

`int sz = suna.size()`
: returns the sizeof `sockaddr_un` part of `sockunixaddr` object `suna`.

`int af = suna.family()`
: returns `sockunixbuf::af_unix` if all is well.

8 sockstream Classes

sockstream classes are designed in such a way that they provide the same interface as their stream counterparts do. We have `isockstream` derived from `istream` and `osockstream` derived from `ostream`. We also have `iosockstream` which is derived from `iostream`.

Each domain also has its own set of `stream` classes. For example, `unix` domain has `isockunix`, `osockunix`, and `iosockunix` derived from `isockstream`, `osockstream`, and `iosockstream` respectively. Similarly, `inet` domain has `isockinet`, `osockinet`, and `iosockinet`.

8.1 iosockstreams

8.1.1 isockstream Class

Since `isockstream` is publicly derived from `istream`, most of the public functions of `istream` are also available in `isockstream`.

`isockstream` redefines `rdbuf()` defined in its virtual base class `ios`. Since, `ios::rdbuf()` is not virtual, care must be taken to call the correct `rdbuf()` through a reference or a pointer to an object of class `isockstream`.

In what follows,

- `sb` is a `sockbuf` object
- `sbp` is a pointer to a `sockbuf` object

`isockstream is(sb)`
: Constructs an `isockstream` object `is` with `sb` as its `sockbuf`.

`isockstream is(sbp)`
: Constructs an `isockstream` object `is` with `*sbp` as its `sockbuf`.

`sbp = is.rdbuf()`
: returns a pointer to the `sockbuf` of the `isockstream` object `is`.

`isockstream::operator -> ()`
: returns a pointer to the `isockstream`'s `sockbuf` so that the user can use `isockstream` object as a `sockbuf` object.

```
is->connect (sa); // same as is.rdbuf()->connect (sa);
```

8.1.2 osockstream Class

Since `osockstream` is publicly derived from `ostream`, most of the public functions of `ostream` are also available in `osockstream`.

`osockstream` redefines `rdbuf()` defined in its virtual base class `ios`. Since, `ios::rdbuf()` is not virtual, care must be taken to call the correct `rdbuf()` through a reference or a pointer to an object of class `osockstream`.

In what follows,

- `sb` is a `sockbuf` object
- `sbp` is a pointer to a `sockbuf` object

`osockstream os(sb)`
: Constructs an `osockstream` object `os` with `sb` as its `sockbuf`.

`osockstream os(sbp)`
: Constructs an `osockstream` object `os` with `*sbp` as its `sockbuf`.

`sbp = os.rdbuf()`
: returns a pointer to the `sockbuf` of the `osockstream` object `os`.

`osockstream::operator -> ()`
: returns a pointer to the `osockstream`'s `sockbuf` so that the user can use `osockstream` object as a `sockbuf` object.

```
os->connect (sa); // same as os.rdbuf()->connect (sa);
```

8.1.3 iosockstream Class

Since `iosockstream` is publicly derived from `ostream`, most of the public functions of `ostream` are also available in `iosockstream`.

`iosockstream` redefines `rdbuf()` defined in its virtual base class `ios`. Since, `ios::rdbuf()` is not virtual, care must be taken to call the correct `rdbuf()` through a reference or a pointer to an object of class `iosockstream`.

In what follows,

- `sb` is a `sockbuf` object
- `sbp` is a pointer to a `sockbuf` object

`iosockstream io(sb)`
: Constructs an `iosockstream` object `io` with `sb` as its `sockbuf`.

`iosockstream io(sbp)`
: Constructs an `iosockstream` object `io` with `*sbp` as its `sockbuf`.

`sbp = io.rdbuf()`
: returns a pointer to the `sockbuf` of the `iosockstream` object `io`.

`iosockstream::operator -> ()`
: returns a pointer to the `iosockstream`'s `sockbuf` so that the user can use `iosockstream` object as a `sockbuf` object.

```
io->connect (sa); // same as io.rdbuf()->connect (sa);
```

8.2 iosockinet Stream Classes

We discus only `isockinet` class here. `osockinet` and `iosockinet` are similar and are left out. However, they are covered in the examples that follow.

8.2.1 isockinet

`isockinet` is used to handle interprocess communication in *inet* domain. It is derived from `isockstream` class and it uses a `sockinetbuf` as its stream buffer. See Section 8.1.3 [iosockstream], page 27, for more details on `isockstream`. See Chapter 4 [sockinetbuf Class], page 14, for information on `sockinetbuf`.

In what follows,

- `ty` is a `sockbuf::type` and must be one of `sockbuf::sock_stream`, `sockbuf::sock_dgram`, `sockbuf::sock_raw`, `sockbuf::sock_rdm`, and `sockbuf::sock_seqpacket`
- `proto` denotes the protocol number and is of type int
- `sb` is a `sockbuf` object and must be in *inet* domain

– `sinp` is a pointer to an object of `sockinetbuf`

`isockinet is (ty, proto)`
: constructs an `isockinet` object `is` whose `sockinetbuf` buffer is of the type `ty` and has the protocol number `proto`. The default protocol number is 0.

`isockinet is (sb)`
: constructs a `isockinet` object `is` whose `sockinetbuf` is `sb`. `sb` must be in *inet* domain.

`isockinet is (sinp)`
: constructs a `isockinet` object `is` whose `sockinetbuf` is `sinp`.

`sinp = is.rdbuf ()`
: returns a pointer to the `sockinetbuf` of `isockinet` object `is`.

`isockinet::operator ->`
: returns `sockinetbuf` of `sockinet` so that the `sockinet` object acts as a smart pointer to `sockinetbuf`.

```
is->localhost (); // same as is.rdbuf ()->localhost ();
```

8.2.2 iosockinet examples

The first pair of examples demonstrates datagram socket connections in the *inet* domain. First, `tdinread` prints its local host and local port on stdout and waits for input in the connection. `tdinwrite` is started with the local host and local port of `tdinread` as arguments. It sends the string "How do ye do!" to `tdinread` which in turn reads the string and prints on its stdout.

```
// tdinread.cc
#include <sockinet.h>

main ()
{
    char buf[256];
    isockinet is (sockbuf::sock_dgram);

    sockinetaddr sina;
    is->bind (sina);

    cout << is->localhost() << ' ' << is->localport() << endl;

    is.getline (buf);
    cout << buf << endl;
}
```

```
// tdinwrite.cc--tdinwrite hostname portno
#include <sockinet.h>

extern "C" int atoi (const char*);

main (int ac, char** av)
{
    osockinet os (sockbuf::sock_dgram);
    sockinetaddr sina (av[1], atoi(av[2]));
    os->connect (sina);
    os << "How do ye do!\n";
}
```

The next example communicates with an nntp server through a `sockbuf::sock_stream` socket connection in *inet* domain. After establishing a connection to the nntp server, it sends a "HELP" command and gets back the HELP message before sending the "QUIT" command.

```
// tnntp.cc
#include <sockinet.h>

main ()
{
    char  buf[1024];
    iosockinet io (sockbuf::sock_stream);
    sockinetaddr sina ("murdoch.acc.virginia.edu", "nntp", "tcp");
    io->connect (sina);
    io.getline (buf, 1024); cout << buf << endl;
    io << "HELP\r\n";
    io.getline (buf, 1024); cout << buf << endl;
    while (io.getline (buf, 1024))
        if (buf[0] == '.' && buf[1] == '\r') break;
        else if (buf[0] == '.' && buf[1] == '.') cout << buf+1 << endl;
        else cout << buf << endl;
    io << "QUIT\r\n";
    io.getline (buf, 1024); cout << buf << endl;
}
```

8.3 iosockunix Classes

We discuss only `isockunix` here. `osockunix` and `iosockunix` are similar.

8.3.1 isockunix class

`isockunix` is used to handle interprocess communication in *unix* domain. It is derived from `isockstream` class and it uses a `sockunixbuf` as its stream buffer. See Section 8.1.3 [iosockstream], page 27, for more details on `isockstream`. See Chapter 6 [sockunixbuf Class], page 21, for information on `sockunixbuf`.

In what follows,

- `ty` is a `sockbuf::type` and must be one of `sockbuf::sock_stream`, `sockbuf::sock_dgram`, `sockbuf::sock_raw`, `sockbuf::sock_rdm`, and `sockbuf::sock_seqpacket`
- `proto` denotes the protocol number and is of type int
- `sb` is a `sockbuf` object and must be in *unix* domain
- `sinp` is a pointer to an object of `sockunixbuf`

`isockunix is (ty, proto)`
: constructs an `isockunix` object `is` whose `sockunixbuf` buffer is of the type `ty` and has the protocol number `proto`. The default protocol number is 0.

`isockunix is (sb)`
: constructs a `isockunix` object `is` whose `sockunixbuf` is `sb`. `sb` must be in *unix* domain.

`isockunix is (sinp)`
: constructs a `isockunix` object `is` whose `sockunixbuf` is `sinp`.

`sinp = is.rdbuf ()`
: returns a pointer to the `sockunixbuf` of `isockunix` object `is`.

`isockunix::operator ->`
: returns `sockunixbuf` of `sockunix` so that the `sockunix` object acts as a smart pointer to `sockunixbuf`.

```
is->localhost (); // same as is.rdbuf ()->localhost ();
```

8.3.2 iosockunix examples

`tsunread` listens for connections. When `tsunwrite` requests connection, `tsunread` accepts it and waits for input. `tsunwrite` sends the string "Hello!!!" to `tsunread`. `tsunread` reads the string sent by `tsunwrite` and prints on its stdout.

```
// tsunread.cc
#include <sockunix.h>
extern "C" int unlink (const char*);
main ()
{
    sockunixbuf sunb (sockbuf::sock_stream);
    unlink ("/tmp/socket+-");
    sockunixaddr suna ("/tmp/socket+-");
    sunb.bind (suna);
    sunb.listen (2);
    isockunix is = sunb.accept ();
    char buf[32];
    is >> buf; cout << buf << endl;
}
```

```
// tsunwrite.cc
#include <sockunix.h>
main ()
{
    osockunix os (sockbuf::sock_stream);
    sockunixaddr suna ("/tmp/socket++");
    os->connect (suna);
    os << "Hello!!!\n";
}
```

9 pipestream Classes

`pipestream` stream classes provide the services of the *UNIX* system calls `pipe` and `socketpair` and the C library function `popen`. `ipipestream`, `opipestream`, and `iopipestream` are obtained by simply deriving from `isockstream`, `osockstream` and `iosockstream` respectively. See Chapter 8 [sockstream Classes], page 26 for details.

In what follows,

- `ip` is an `ipipestream` object
- `op` is an `opipestream` object
- `iop` is an `iopipestream` object
- `cmd` is a char* denoting an executable like "wc"
- `ty` is of type `sockbuf::type` indicating the type of the connection
- `proto` is an `int` denoting a protocol number

`ipipestream ip(cmd)`
: construct an `ipipestream` object `ip` such that the output of the command `cmd` is available as input through `ip`.

`opipestream op(cmd)`
: construct an `opipestream` object `op` such that the input for the command `cmd` can be send through `op`.

`iopipestream iop(cmd)`
: construct an `iopipestream` object `iop` such that the input and the output to the command `cmd` can be sent and received through `iop`.

`iopipestream iop(ty, proto)`
: construct a `iopipestream` object `iop` whose socket is a socketpair of type `ty` with protocol number `proto`. `ty` defaults to `sockbuf::sock_stream` and `proto` defaults to 0. Object `iop` can be used either as a `pipe` or as a `socketpair`.

`iop.pid ()`
: return the process id of the child if the current process is the parent or return 0. If the process has not forked yet, return -1.

`iopipestream::fork ()`
: `fork()` is a static function of class `iopipestream`. `fork()` forks the current process and appropriately sets the `cpid` field of the `iopipestream` objects that have not forked yet.

9.1 pipestream as pipe

`pipe` is used to communicate between parent and child processes in the *unix* domain.

The following example illustrates how to use `iopipestream` class as a `pipe`. The parent sends the string "I am the parent" to the child and receives the string "I am the child" from child. The child, in turn, receives the string "I am the parent" from parent and sends the string "I am the child" to the parent. Note the same `iopipestream` object is used for input and output in each process.

```
#include <pipestream.h>

main()
{
        iopipestream p;
        if ( p.fork() ) {
                char buf[128];
                p << "I am the parent\n";
                cout << "parent: ";
                while(p >> buf)
                        cout << buf << ' ';
                cout << endl;
        }else {
                char buf[128];
                p.getline(buf, 127);
                cout << "child: " << buf << endl;
                p << "I am the child\n";
        }
}
```

9.2 pipestream as socketpair

Like pipes, socketpairs also allow communication between parent and child processes. But socketpairs are more flexible than pipes in the sense that they let the users choose the socket type and protocol.

The following example illustrates the use of `iopipestream` class as a `socketpair` whose type is `sockbuf::sock_dgram`. The parent sends the string "I am the parent" to the child and receives the string "I am the child" from the child. The child, in turn, receives and sends the strings "I am the parent" and "I am the child" respectively from and to the parent. Note in the following example that the same `iopipestream` object is used for both the input and the output in each process.

```
#include <pipestream.h>

main()
{
	iopipestream p(sockbuf::sock_dgram);
	if ( iopipestream::fork() ) {
		char buf[128];
		p << "I am the parent\n";
		p.getline(buf, 127);
		cout << "parent: " << buf << endl;
	}else {
		char buf[128];
		p.getline(buf, 127);
		cout << "child: " << buf << endl;
		p << "I am the child\n";
	}
}
```

9.3 pipestream as popen

`popen` is used to call an executable and send inputs and outputs to that executable. For example, the following function executes "/bin/date", gets its output, and prints it to stdout.

```
#include <pipestream.h>

main ()
{
    char buf[128];
    ipipestream p("/bin/date");

    p.getline (buf, 127);
    cout << buf << endl;
}
```

Here is an example that prints "Hello World!!" on stdout. It uses `opipestream` object.

```
#include <pipestream.h>

main ()
{
    opipestream p("/bin/cat");
    p << "Hello World!!\n";
}
```

The following example illustrates the use of `iopipestream` for both input and output.

```
#include <pipestream.h>

main()
{
        char buf[128];
        iopipestream p("lpc");
        p << "help\nquit\n";
        while ( p.getline(buf, 127) ) cout << buf << endl;
}
```

10 Error Handling

Each class in the Socket++ library uses `error(const char*)` member function to report any errors that may occur during a system call. It first calls `perror()` to report the error message for the `errno` set by the system call. It then calls `(*lib_error_handler) (const char* nm, const char* errmsg)` where nm is the name of the class.

If the user wants to handle the error more gracefully he can define his own error handler instead of the default `lib_error_handler`.

In what follows,

- `eh` is of type void (*)(const char*, const char*)

`set_lib_error_handler(eh)`

returns the old lib error handler after setting `lib_error_handler` to eh. This function is part of `libg++` library and is not part of Socket++ library.

11 Pitfalls

Deadlocks in datagram sockets are the most common mistakes that novices make. To alleviate the problem, `sockbuf` class provides timeout facilities that can be used effectively to avoid deadlocks.

Consider the following simple tsmtp example which sends the HELP command to a smtp server and gets back the help message. Suppose it does not know the size of the help message nor the format of the message. In such cases, the timeout facilities of `sockbuf` class provides the required tools.

The example terminates the help message reception if the there is no input activity from the smtp server for 10 seconds.

tsmtp.cc

```
#include <sockinet.h>

main()
{
sockinetaddr sina("kelvin.seas.virginia.edu", "smtp", "tcp");
iosockinet   sio(sockbuf::sock_stream);

sio->connect(sina);

char buf[512];
sio.getline(buf, 511); cout << buf << endl;
sio << "HELO kelvin\n";
sio.getline(buf, 511); cout << buf << endl;

sio << "HELP\n";

        // set the receive timeout to 10 seconds
        int tmo = sio->recvtimeout(10);

while ( sio.getline(buf, 511) ) cout << buf << endl;
// if the above while loop terminated due to timeout
// clear the state of sio.
if ( !sio->is_eof() )
sio.clear();
sio->recvtimeout(tmo); // reset the receive timeout time

sio << "QUIT\n";
```

```
sio.getline(buf, 511); cout << buf << endl;
}
```

Index

T

U

W

Table of Contents