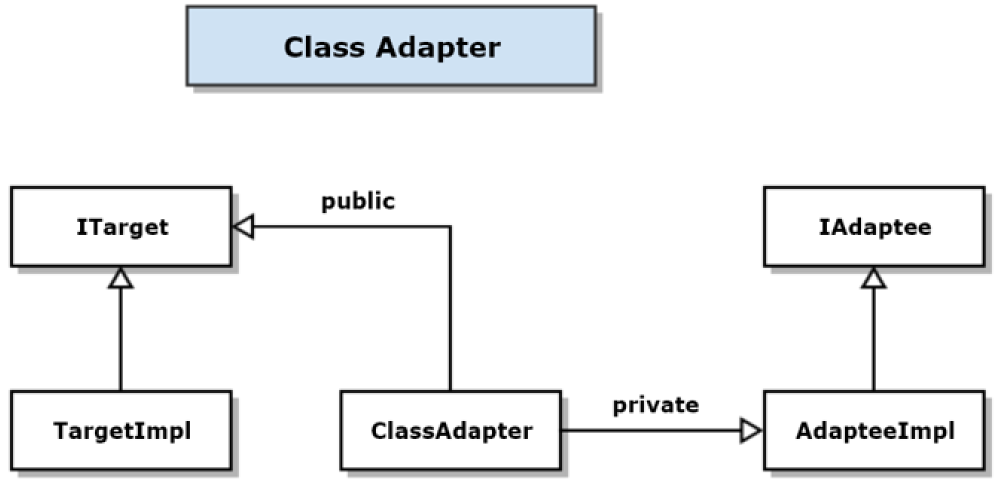
Adapter Application Code

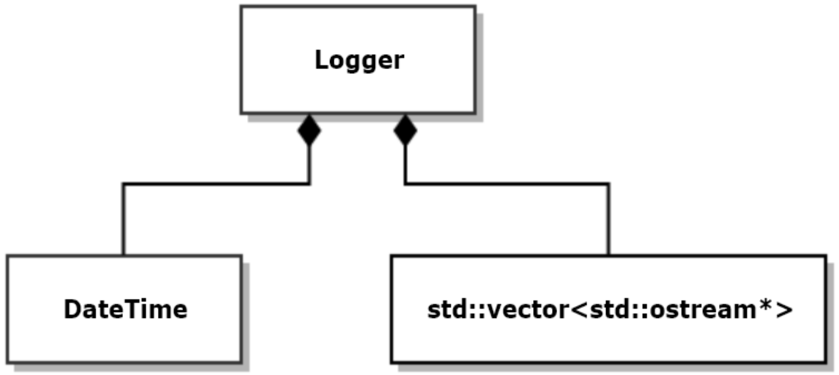
Jim Fawcett

Design Patterns, Fall 2018

Application Specific Class Adapter

- Adapts Msg-Passing Comm to std::ostream interface
- Use in logger
  - Uses multiple streams
  - This adaption lets it log to other processes or machines using std::ostream interface, i.e., operator<< or write



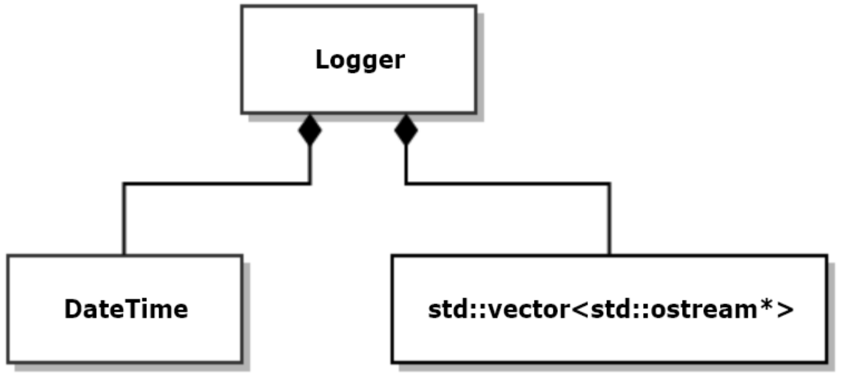


Logging Application

- Accepts multiple streams that implement std::ostream interface
- Will adapt Msg-Passing Comm to use std::ostream interface
- Can then log from one process to another

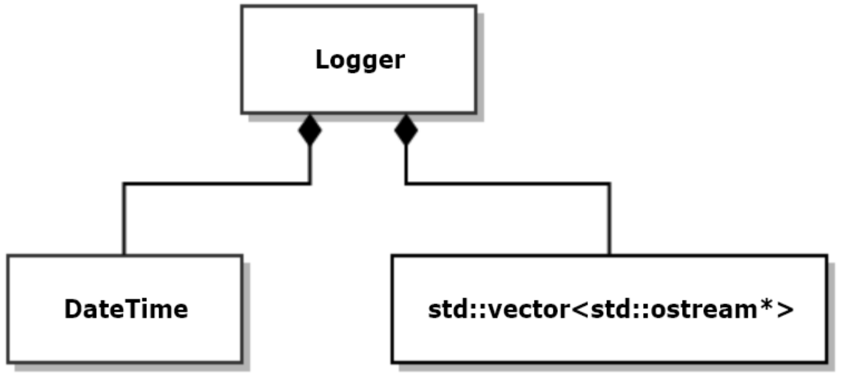
Singleton Logger supports multiple streams

```
class Logger
 {
   using Streams = std::vector<std::ostream*>;
   using Terminator = std::string;

 public:
   void addStream(std::ostream* pStream)
   {
     streams_.push_back(pStream);
   }
   bool removeStream(std::ostream* pStream)
   {
     Streams::iterator iter = std::find(streams_.begin(), streams_.end(), pStream);
     if (iter != streams_.end())
     {
       streams_.erase(iter);
       return true;
     }
     return false;
   }
```



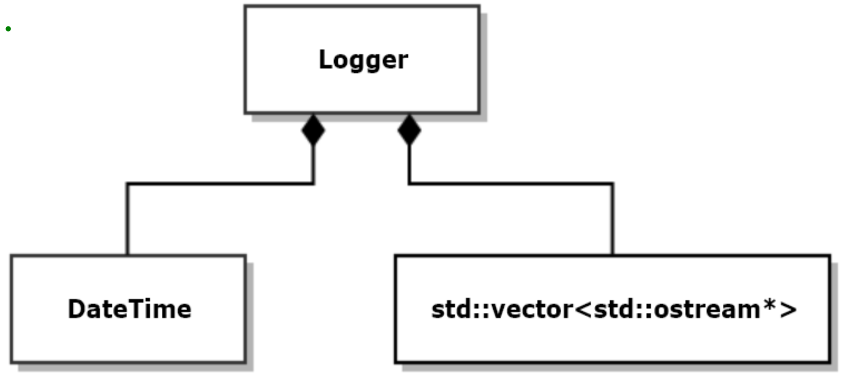
Singleton Logger supports multiple streams

```
void writeHead(const std::string& msg)
   {
     for (auto pStrm_ : streams_)
     {
       *pStrm_ << msg.c_str() << " : ";
       *pStrm_ << DateTime().now() << trm_.c_str();
     }
   }
   void write(const std::string& text)
   {
     for (auto pStrm_ : streams_)
       *pStrm_ << text.c_str() << trm_.c_str();
   }
   void writeTail(const std::string& msg = "end of log")
   {
     for (auto pStrm_ : streams_)
       *pStrm_ << msg.c_str();
   }
```

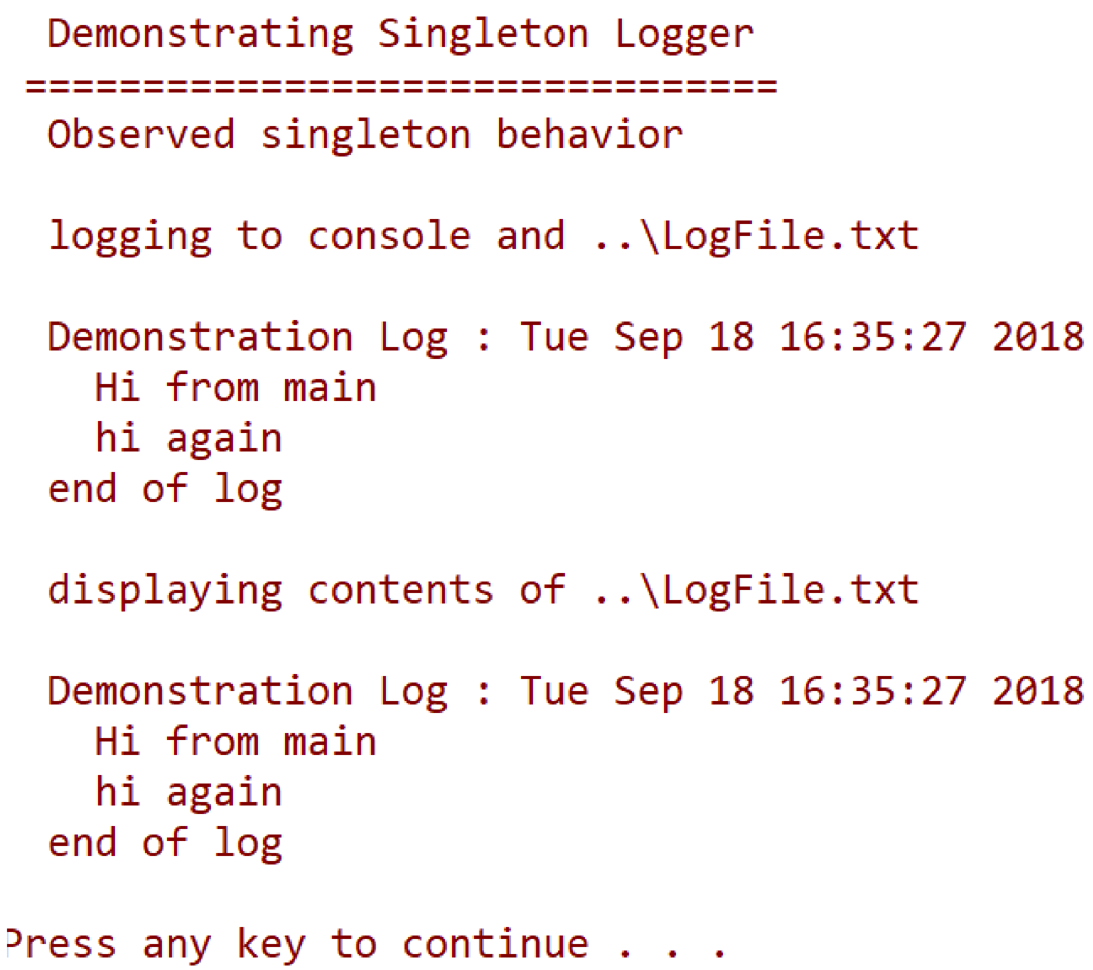


Singleton Logger

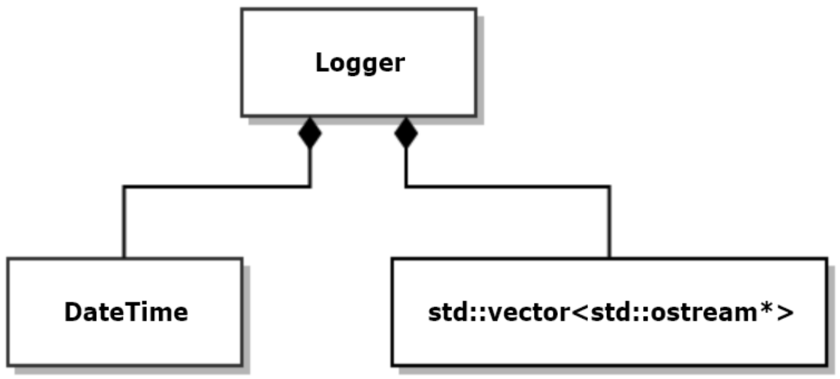
```
// Thread-safe singleton access:
   // - Does not attempt to improve performance by double-check locking
   // - That may fail occasionally, in C++, due to caching of instance_.
   // - Since accesses are rare, usually only a very few times per
   //   execution, performance degradation is very small.

   static Logger* getInstance()
   {
     std::lock_guard<std::mutex> lck(mtx);
     if (instance_ == nullptr)
     {
       instance_ = new Logger;
     }
     return instance_;
   }

   Logger(const Logger&) = delete;
   Logger& operator=(const Logger&) = delete;

 private:
   Logger()
   {
     addStream(&std::cout);
   }
   static Logger* instance_;
   static std::mutex mtx;
   Streams streams_;
```



Singleton Logger Demo Output



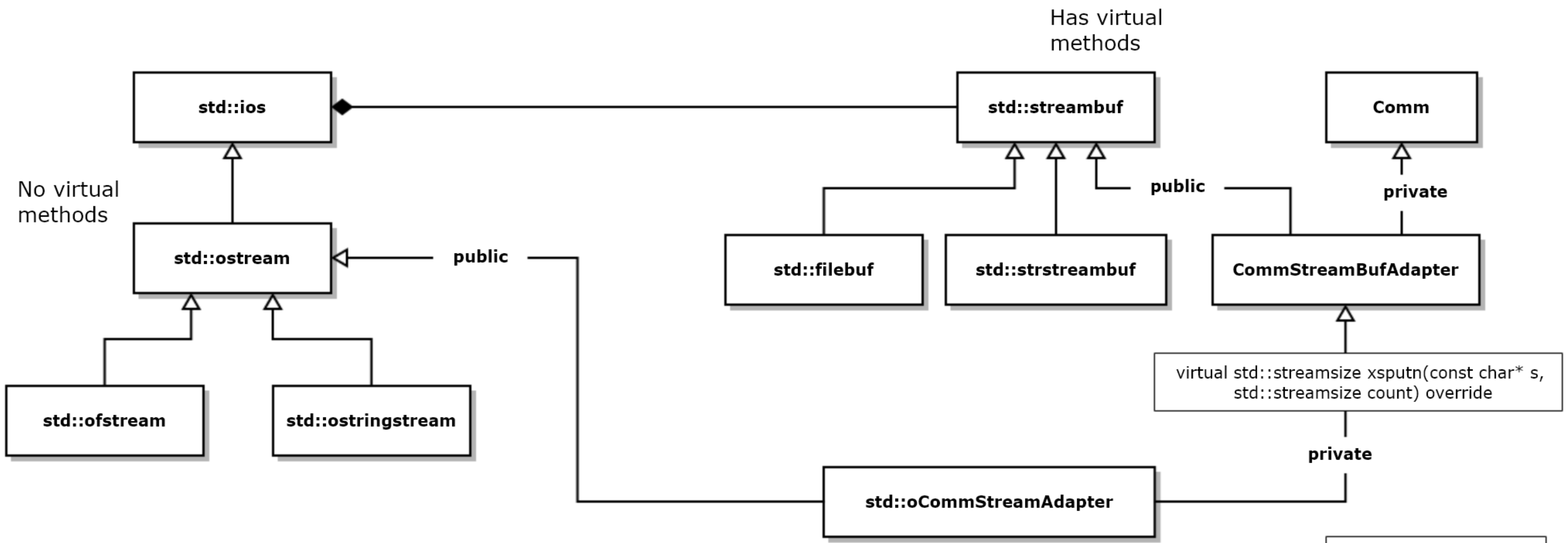
```
 Demonstrating Singleton Logger
==================================
 Observed singleton behavior

 logging to console and ..\LogFile.txt

 Demonstration Log : Tue Sep 18 16:35:27 2018
   Hi from main
   hi again
 end of log

 displaying contents of ..\LogFile.txt

 Demonstration Log : Tue Sep 18 16:35:27 2018
   Hi from main
   hi again
 end of log

ress any key to continue . . .
```



Adapt Msg-Passing Comm

- Adapt Comm to std::ostream
- Can then log to another process, using ostream interface, but passing messages behind the curtain.
- One hurdle:
  - std::ostream doesn’t have any virtual functions to override
  - std::streambuf to the rescue. It has virtual methods and std::ostream is just a wrapper that uses the std::streambuf for all the real work.
  - So, we adapt std::streambuf.

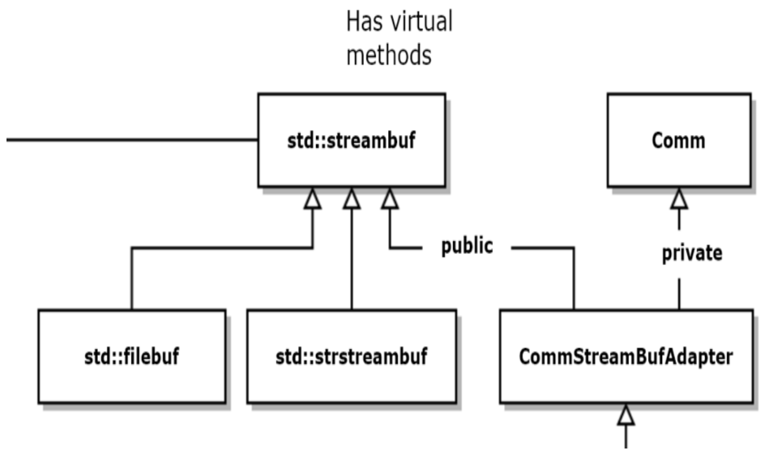
Stream Adapter for Comm

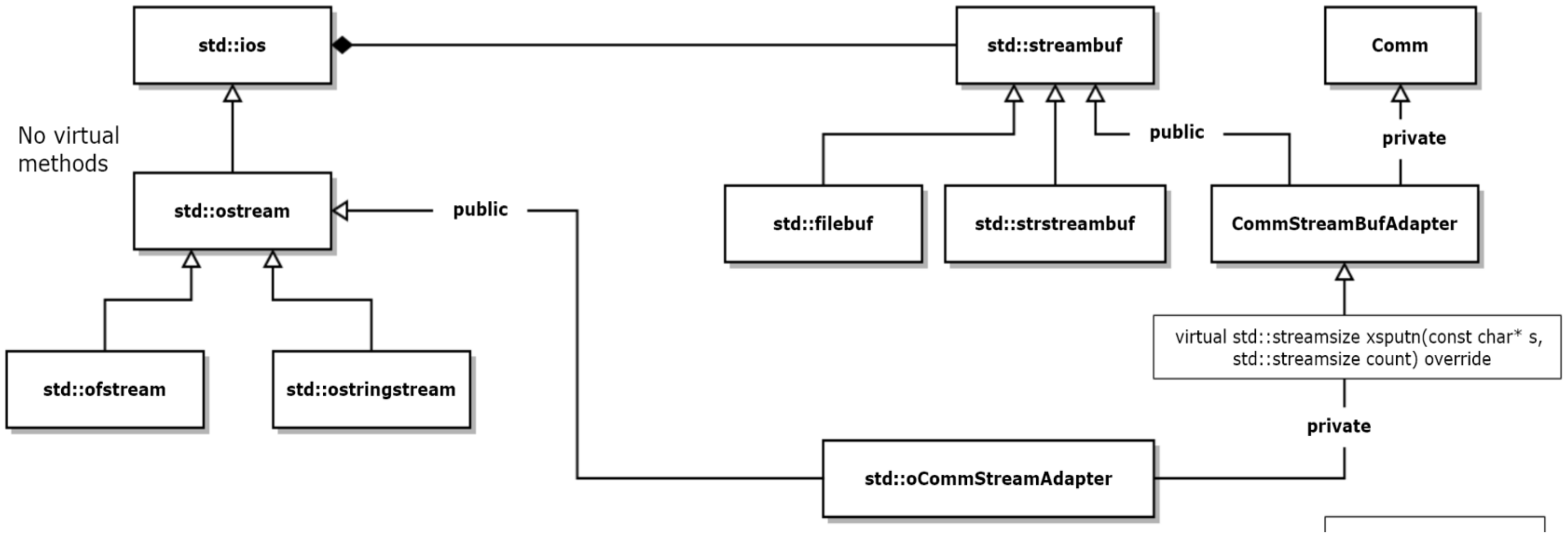


Build oCommStreamAdapter using CommStreamBuffAdapter

```
class oCommStreamAdapter : private CommStreamBufAdapter, public std::ostream
{
public:
  oCommStreamAdapter(EndPoint to, EndPoint from)
    : CommStreamBuf(to, from), std::ostream((CommStreamBufAdapter*)this) {}

  void close() { closeComm(); }
private:
  SocketSystem ss;  // declaration needed for Comm's Socket Library
};
}
```

```cpp
///////////////////////////////////////////////////////////////////
 // CommStreamBufAdapter class
 // - class adapter
 // - adapts Comm to act like a std::streambuf

 class CommStreamBufAdapter : public std::streambuf, private Comm
 {
 public:
   CommStreamBufAdapter(EndPoint to, EndPoint from) : to_(to), from_(from), Comm(from)
   {
     start();  // start local comm running
   }
   virtual ~CommStreamBufAdapter() {}

   virtual std::streamsize xsputn(const char* s, std::streamsize count) override
   {
     // xsputn accepts characters from any of the ostream (non-virtual) methods

     Message msg = makeMessage(to_, from_, s);  // make message using stream chars
     postMessage(msg);                          // post it to Comm
     return count;
   }
   void closeComm() { stop(); }
 private:
   EndPoint to_;
   EndPoint from_;
 };
```





```
//////////////////////////////////////////////////////////////////////
 // oCommStreamAdapter class
 // - uses adapted streambuf in its internal inherited ostream

 class oCommStreamAdapter : private CommStreamBufAdapter, public std::ostream
 {
 public:
   oCommStreamAdapter(EndPoint to, EndPoint from)
     : CommStreamBufAdapter(to, from), std::ostream((CommStreamBufAdapter*)this) {}

   void close() { closeComm(); }
 private:
   SocketSystem ss;  // declaration needed for Comm's Socket Library
 };
```

That’s It!

- Just a tiny bit of code to adapt Comm to std::ostream.
- Comm does all the heavy TCP work.
- std::ostream handles writes and insertions using our adapted std::streambuf
- Piece of Cake!

Sender's main

```
int main()
{
  std::cout << "\n  Demonstrating oCommStreamAdapter Sender";
  std::cout << "\n =========================================";

  EndPoint to("localhost", 8080);
  EndPoint from("localhost", 8081);
  oCommStreamAdapter sa(to, from);

  // use ostream operator<<
  std::string firstMsg = "hi from client";
  std::cout << "\n  sending 1st message \"" << firstMsg << "\" to receiver process";
  sa << firstMsg;

  // use ostream write method
  std::string secondMsg = "hi again from client";
  std::cout << "\n  sending 2nd message \"" << secondMsg << "\" to receiver process";
  sa.write(secondMsg.c_str(), secondMsg.length());

  // tell receiver to shut down
  std::string thirdMsg = "quit";
  std::cout << "\n  requesting receiver process to shutdown with \"" << thirdMsg << "\"";
  sa << thirdMsg;

  std::cout << "\n\n";
  std::cout << "\n  press key to quit\n";
```

Configure oCommStreamAdapter for a specified channel.

Using std::ostream insertion

Using std::ostream::write

Receiver’s main

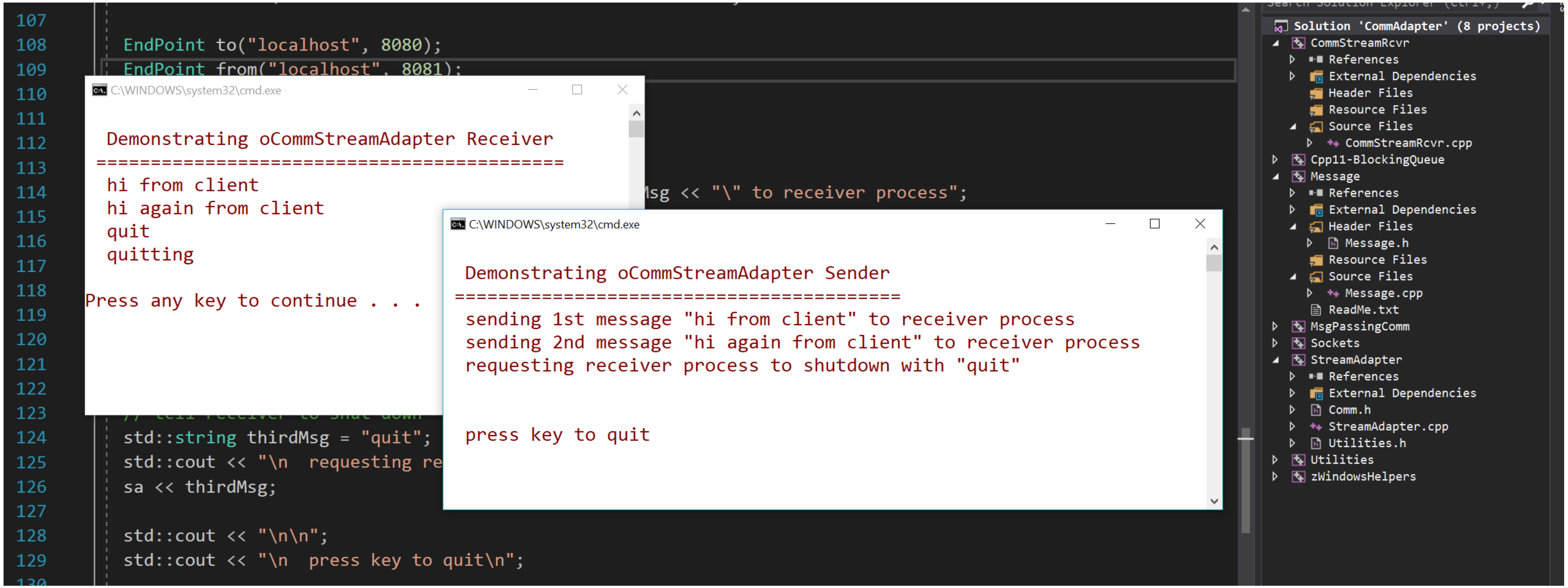
```
int main()
{
  std::cout << "\n  Demonstrating oCommStreamAdapter Receiver";
  std::cout << "\n ===========================================";

  SocketSystem ss;
  EndPoint ep("localhost", 8080);
  Comm comm(ep, "testComm");
  comm.start();

  while (true)
  {
    Message rcvd = comm.getMessage();
    //rcvd.show();
    if (rcvd.containsKey("content"))
    {
      std::string value = rcvd.attributes()["content"];
      std::cout << "\n  " << value;
      if (value == "quit")
      {
        break;
      }
    }
  }
  std::cout << "\n  quitting";
  comm.stop();
```

Receiver’s comm is started

Receiver displays text it got from comm message

comm is closed below

Sender process logging to Receiver process



```
EndPoint to("localhost", 8080);
EndPoint from("localhost", 8081);
```

```
sg << "\" to receiver process";
```

```
std::string thirdMsg = "quit";
std::cout << "\n  requesting re
sa << thirdMsg;

std::cout << "\n\n";
std::cout << "\n  press key to quit\n";
```

C:\WINDOWS\system32\cmd.exe

```
 Demonstrating oCommStreamAdapter Receiver
===========================================
 hi from client
 hi again from client
 quit
 quitting

Press any key to continue . . .
```

C:\WINDOWS\system32\cmd.exe

```
 Demonstrating oCommStreamAdapter Sender
=========================================
 sending 1st message "hi from client" to receiver process
 sending 2nd message "hi again from client" to receiver process
 requesting receiver process to shutdown with "quit"


 press key to quit
```

That’s All Folks!