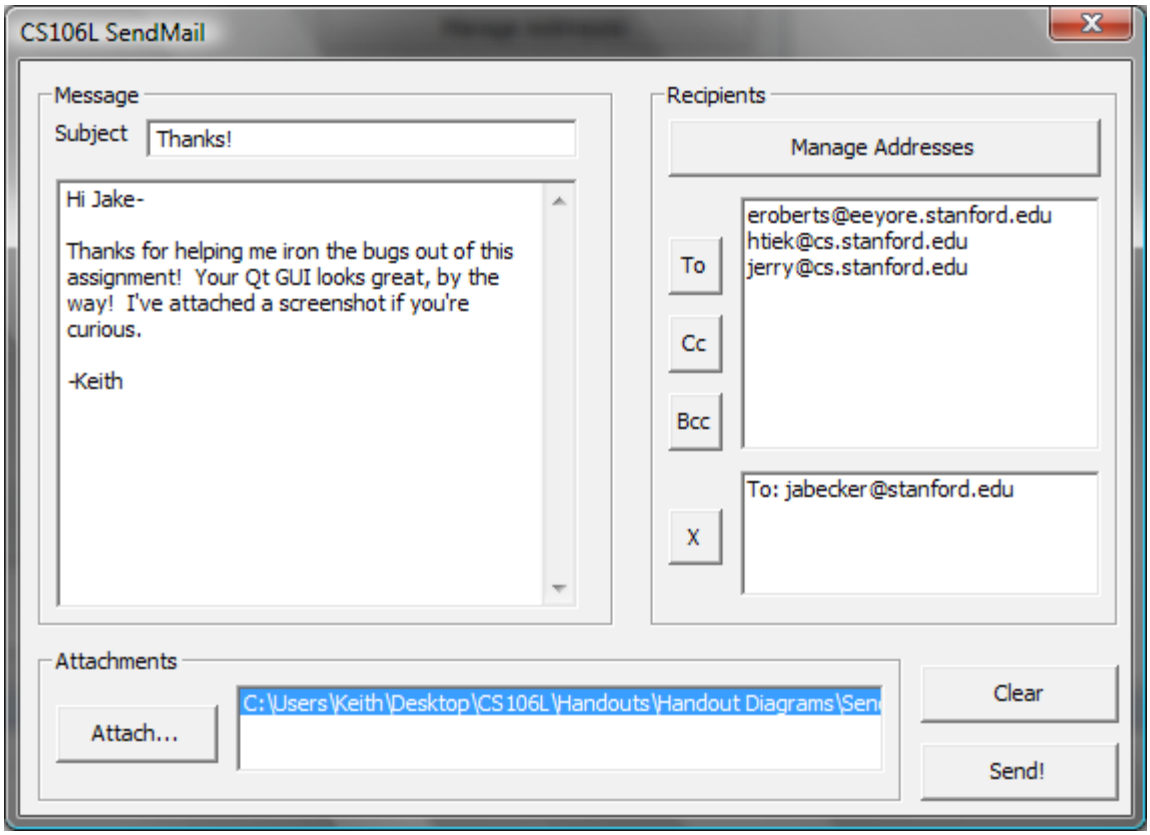
Assignment 0: SendMail*

Due April 22, 11:59 PM

Introduction

Over the past two weeks we've explored the streams library and parts of the STL. It's now time to synthesize your C++ skills and a few new concepts to build *SendMail*, a program that can send email messages to your friends and family. SendMail is a minimal email client that maintains an address book and sends emails to recipients in that address book. Here's a screenshot of SendMail running in Windows:



On the left side of this window is a panel where the user can input the subject and the actual message text. On the right is the address book listing all of the user's addresses, as well as a list of recipients (in this case, jabecker@stanford.edu). The bottom of the window lists all of the email attachments, and gives the user the option to send the email.

In this assignment, we won't ask you to do any windowed programming or GUI design. C++ does not have a standard windowing library, and C++ programs that use GUIs must rely on third-party libraries. The Windows version of the SendMail program uses Microsoft's Win32 API for its windowing, while the Mac and Linux versions of the program are based on the popular Qt framework (Qt is pronounced "cute"). Since we have not talked about either of these libraries in class, we aren't expecting you to work

* This assignment would not have been possible if not for the generous help of Jake Becker, Feross Aboukhadijeh, and Riddhi Mittal.

with them, and accordingly we've provided you with starter code that handles all of the windowing. Your assignment is to implement the *backend* of the SendMail program. You will be provided a working GUI and will implement the code for maintaining the address book and actually sending emails to their intended recipients. In the course of doing so, you will

- Gain experience with the streams library and basic STL container classes.
- See how SMTP works and how email clients send messages.
- Get a taste of networking and socket programming.
- Learn how to read class documentation in C++ projects.

Although this assignment may initially seem intimidating, the actual amount of code you will need to write for this program is rather small. It is entirely possible to write a solution in under two hundred lines of code, and under one hundred if you're particularly concise. The rest of this handout describes the necessary background material you will need in order to implement a networked mail client, along with a breakdown of what tasks you will need to perform.

How Email Works

Before detailing exactly what you're responsible for, let's take a quick detour to understand how emails are sent. When you write an email using your favorite program and click the "send" button, your message is sent to a *mail server*, a special computer designed to transmit mail from senders to recipients. Upon receipt of your message, the mail server forwards your message to other mail servers until it eventually arrives at its destination. In other words, to get your message to your recipients, you need only give that message to the mail server; the server will handle the rest.

When computers communicate with one another, they usually do so according to some *protocol*. For example, the hypertext transfer protocol (HTTP) defines a specific system by which a computer can download a webpage from a server, while the session initiation protocol (SIP) provides a standardized means for web phones to call one another. To send email messages to mail servers, computers use *SMTP*, the *Simple Mail Transfer Protocol*. SMTP is a formal specification of how a mail client communicates with a mail server, and provided that both parties implement the protocol correctly the client will be able to transmit her message to the server. Here is a sample conversation between a mail client and mail server using SMTP, in which the user (me) sends an email to two recipients:

Server:	220 Mail server at your service!
Client:	**HELO 128.12.124.43**
Server:	250 server266.com
Client:	**MAIL FROM: htiek@cs.stanford.edu**
Server:	250 Okay
Client:	**RCPT TO: jabecker@stanford.edu**
Server:	250 Okay
Client:	**RCPT TO: feross@stanford.edu**
Server:	250 Okay
Client:	**DATA**
Server:	354 Go Ahead
Client:	**From: Keith Schwarz <htiek@cs.stanford.edu>** **To: Jake Becker <jabecker@stanford.edu>** **Cc: Feross Aboukhadijeh <feross@stanford.edu>** **Subject: Sample Email Message** **Hey all-** **This is a sample email message.** **-Keith** **.**
Server:	250 Message Received! Will send.
Client:	**QUIT**
Server:	221 Have a nice day!

The text in **bold** represents messages my computer sent to the mail server, and the lighter text indicates messages from the server to my computer. Don't worry if you don't understand the above transcript right now. By the time you've finished the next few sections, you should be able to read this conversation with minimal effort.

Each SMTP session reads more or less like a script. Each of the computers (the client and the SMTP server) speak a line in turn, beginning with the server. However, the sorts of messages sent by each computer are markedly different. The client's messages are *requests* to the server that ask the server to execute a particular command, while the server's messages are *responses* indicating the result of that command. Let's take a look at a sample request from the client; in particular, the second message sent by the client. This message is:

MAIL FROM: htiek@cs.stanford.edu

This command tells the server that I want to send an email from the account htiek@cs.stanford.edu. The server processed this command, then sent back the following information:

250 Okay

Notice that this reply consists of a three-digit number (called a *reply code*), along with the message "Okay." Whenever the server sends a message back to the client about the last request, the response will always begin with a a three-digit reply code like this one. Reply codes allow the client to quickly glean what happened during the last transmission, and in fact it is possible to hold an entire SMTP conversation without ever looking at any part of the server's response other than the reply code. All information contained in a server response beyond the reply code is purely informational and has no relevance to the protocol's execution. This means that the server response

```
354 Go Ahead
```

is entirely equivalent to this one

```
354 Send away!
```

because the reply code is the same. However, it is **not** the same as

```
250 Go Ahead
```

since the reply code of this message (250) does not match that of the original message (354).

The client of an SMTP session is always responsible for driving the conversation, but must do so within a fairly rigid framework. The structure, timing, and content of the client's commands are well-specified, and any deviation from the expected format will confuse the server and cause the SMTP session to fail. Specifically, an SMTP session consists of several steps which must be followed in order. To successfully send an email as an SMTP client, you should perform the following steps in the following order:

- **Establish the Connection**. To begin an SMTP session, you first need to establish a connection with an SMTP server. As soon as the connection is made, the server will send back a response containing a 220 reply code to let you know that the connection is valid. In your implementation of the SMTP session, you should be sure to check for proper receipt of this message. If you do not receive this message, or if you receive a message with a different reply code, then the server will not be able to process your email.

  Example server-to-client message: `220 Email Server Ready`

- **Say Hello**. Once you have connected to the server and verified that the connection is valid, you must identify yourself to the mail server by sending a HELO. The structure of a HELO message is **HELO *name***, where ***name*** is some string identifying you (though not your email address). The ***name*** parameter is usually the hostname of the computer sending the email, but you can use whatever you'd like. In my solution, I used "CS106L," and I doubt that doing so in your solution will cause any problems.

  Upon receipt of a HELO message, the server usually responds with a 250 reply code and an informational message containing its name. Be sure to report an error if you do not receive a 250 reply code, since this would indicate that the server does not want to talk to you.

  Example exchange:
  Client: `HELO Mail Client`
  Server: `250 smtp.stanford.edu`

- **Identify Yourself**. Your next task is to indicate which email address to send the email from. This is done using the MAIL FROM command, which has the syntax **MAIL FROM: *email-address***, where ***email-address*** is your email address (for example, htiek@cs.stanford.edu). The mail server then responds with a 250 status code if it will receive a message from that sender. Interestingly, the identity of the sender is not authenticated. In fact, it is possible for the sender to completely lie about her email address. This is called *email spoofing* and is one of the reasons that spam is so rampant across the Internet. *Please do not use this assignment to spoof email addresses*.

  Example exchange:
  Client: **`MAIL FROM: htiek@cs.stanford.edu`**
  Server: **`250 Okay`**

- **Identify the Recipients**. At this point, you must specify who you want to send the message to. For each of the message's recipients, send a RCPT TO message telling the server who that recipient is. The RCPT TO message has the syntax **RCPT TO: *email-address***, where ***email-address*** is the recipient's email address. Note that the command is *always* RCPT TO, even if the recipient is Cc'ed or Bcc'ed on the email.* The server will respond with a 250 reply code after each recipient, and you should report an error if any of the recipients are rejected by the server (e.g. you receive a reply other than 250).

  Example exchange:
  Client: **`RCPT TO: jabecker@stanford.edu`**
  Server: **`250 Okay`**
  Client: **`RCPT TO: feross@stanford.edu`**
  Server: **`250 Okay`**

- **Send the Message**. Now that the sender and recipient are known, you are ready to transmit the email message. This is done using the DATA command. Unlike the commands we've seen so far, the DATA command does not contain the actual email. Instead, the client sends the DATA command by itself, and the server responds with a 354 message indicating that it is ready to receive. The client then sends the contents of the email to the server, then sends a newline, a period, and another newline. All of the message contents up to the newline/dot/newline triple are treated as the email message, and the server sends back a 250 status message if all went well.

  One detail we've glossed over so far is how the email message is encoded when it's sent to the server. Email messages are encoded using a system called *MIME* (Multipurpose Internet Mail Extensions). While the structure and content of MIME-formated emails is a fairly interesting topic, we won't address it in this assignment. You will be provided with a class that automatically generates MIME-formatted email messages, and so the focus of this assignment will be implementing the SMTP exchange. If you are curious what a MIME message looks like, you may want to peek around the contents of the `EmailMessage.cpp` file included with your project.

---

* Email recipients can still tell what type of recipient they are because the body of the email will contain a header with this information. In fact, most modern email systems will only use information from the email message itself when displaying the subject, sender, recipient, etc.

Example exchange:
Client: **`DATA`**
Server: **`354 Go Ahead`**
Client: *(body of the email)*
Client: **`.`**
Server: **`250 Message Received`**

- **Say Goodbye**. Finally, once the email body has been transmitted, you should tell the server that you are done and close the connection. This is done by sending the QUIT command, at which point the server should send a 221 message and close the connection. No further action is required on your part.

  Example exchange:
  Client: **`QUIT`**
  Server: **`221 Have a nice day!`**

Network Connections

In order to call up an SMTP server to send an email message, you will need to establish a network connection to that computer. When contacting a remote machine, you need two pieces of information. First, you need the *hostname* of that machine, which can either be an IP address (e.g. 137.42.27.17) or a string (e.g. www.google.com). For this assignment, you will be contacting Stanford's SMTP server, which is at

`smtp.stanford.edu`

Additionally, you will need a *port number*. A single computer might act as a server for multiple protocols; for example, a computer could be a web server, SMTP server, and FTP server. In order to ensure that requests to that server are interpreted properly, all network traffic addressed to a computer specifies which *port* that message should be received on. A port is a logical (not physical) destination that simply allows the computer to figure out what program a message is intended for. For example, web traffic is typically done on port 80 (or port 443 for a secure connection), while FTP traffic is on ports 20 and 21. SMTP servers listen in on port 25, and so when establishing a connection to the Stanford SMTP server you will need to make sure that you connect to port 25. Otherwise, your connection will be refused and the server will not accept your email. As an FYI, the common term for a pair of an address and a port is a *socket*, and you will undoubtedly hear this term later in your programming career.

In this programming assignment, we will provide you a stream class that can connect to a mail server and transmit data. More information about that class is provided later in this handout.

The Assignment

Your assignment is to implement the four functions contained in the `mail.h` header file packaged with the starter code. These functions are invoked by the GUI in response to particular events, so when you implement these functions the SendMail program will automatically begin using them. The functions in the `mail.h` header manage the address book and send email messages via SMTP, and should be reasonably-well documented. Please please your implementations in the `mail.cpp` file; when testing your submission, we will only be looking at this file unless you explicitly mention otherwise. Also note that because the GUI code already calls into the functions from `mail.h`, you should not change the prototypes of any of the functions defined there, since this will break the provided GUI code.

Because this project relies on third-party libraries, setting up this project for compilation is slightly more difficult than usual. Before working on this assignment, follow the directions relevant for your operating system:

- **Getting Started on Windows**. If you're using Windows, you should have no trouble getting this project set up. Download the Windows starter files from the CS106L website, unzip the project directory, and open the project file in Visual Studio. Compile and run the program to verify that everything works correctly. You should see the compose window, but the address book window will be empty. If you've reached this point, you're all set to go and can start writing code. If not, let me know and we can try to diagnose what's going wrong.

- **Getting Started on Mac OS X**. The implementation of the GUI for Mac OS X uses a set of windowing libraries called Qt (pronounced "cute"). Although these libraries are written in C++, they are not part of the C++ standard and most compilers, including Xcode, do not ship with them installed. In order to compile and run this assignment, you will need to download and install the Qt libraries. To do this, go to

  http://qt.nokia.com/downloads/mac-os-cpp

  The download should begin automatically. Once it completes, extract and run the installer. The installer will need about 500MB of hard disk space; contact me if this is a problem on your system. You should be perfectly fine using all of the default settings during installation. After installation completes, download the Mac OS X starter code from the CS106L website and unzip the project directory. Inside the folder, you should find a program called **setup**. This program will create an Xcode project for you, and you should only need to run it once. Run the program, then open up the generated project file in Xcode. You should be all set at this point. Note that you only need to do this complicated setup once; after you've created the Xcode project file, you can treat it like a regular Xcode project from here on out.

  To verify that everything installed correctly, run the program from Xcode. You should see the compose window, but the address book window will be empty. If you've reached this point, you're all set to go and can start writing code. If not, let me know and we can try to diagnose what's going wrong.

  Note: Unfortunately, the Stanford cluster computers do not have Qt installed and because student accounts are not administrators, you cannot set up this project on the cluster Macs. If you want to use the cluster machines, you will need to use the Windows version of this project.

- **Getting Started on Linux**. The implementation of the GUI for Mac OS X uses a set of windowing libraries called Qt (pronounced "cute"). Although these libraries are written in C++, they are not part of the C++ standard and most compilers, including g++, do not ship with them installed. In order to compile and run this assignment, you will need to download and install the Qt libraries. To do this, open a terminal, then execute the command

  ```
  sudo apt-get install libqt4-dev
  ```

  Once you've installed Qt, download the Linux starter files from the CS106L website and extract the files to the directory of your choice. Unlike traditional C++ projects, programs written using Qt have a two-step build process. First, execute the `qmake` command to have Qt set up the windowing objects, and then execute `make` as you normally would to build the executable. You do

not need to run `qmake` multiple times, though if you execute a `make clean` you will need to re-run `qmake` to build the intermediary files.

After following these directions, compile the program with `make` and run the program by executing `./email`. You should see the compose window, but the address book window will be empty. If you've reached this point, you're all set to go and can start writing code. If not, let me know and we can try to diagnose what's going wrong.

Note: Unfortunately, the Stanford cluster computers do not have Qt installed and because student accounts are not administrators, you cannot set up this project in Linux on the cluster computers. If you want to use the cluster machines, you will need to use the Windows version of this project.

While you are free to implement this assignment in any way you see fit, some of the functions are easier to implement than others and I've broken the project down into three smaller tasks. If you finish the tasks in this order, you will complete the assignment with minimal hassle.

Step 0: Implement `LoadAddressBook` and `SaveAddressBook`

One of SendMail's main tasks is managing an address book containing a list of email addresses. For simplicity, the address book is represented by a `vector<string>`. The `mail.h` header contains two functions, `SaveAddressBook` and `LoadAddressBook`, which are responsible for saving and loading the user's address book to and from disk. Your first task is to implement these two functions. There are many correct ways to do so, and you are free to choose any implementation that works (provided, of course, that they adhere to the interface specified by `mail.h`).

The starter code does not contain an address book file, and so it is up to you to name and design the format of this file. You might store the email addresses in a file with one address per line, or separated by some special character that cannot normally appear in email addresses. If you'd like, you might store the number of addresses as the first line of the file, or alternatively you might not store the number anywhere in the file.

Step 1: Learn `nstream` and `EmailMessage`

You now have a working address book, and it's almost time to start sending emails. However, before you can transmit email messages, you will need to familiarize yourself with two custom classes – `EmailMessage`, which stores an email message, and `nstream`, a stream object that will let you communicate with Stanford's SMTP server.

`EmailMessage` represents an email message. It stores the email addresses of the sender and the recipients, the full text of the message body, and a list of attachments to that email. Additionally, it exports a function that yields a MIME representation of that message which can be sent to the SMTP server. The comments in the `EmailMessage.h` header file should give you a good sense of how to query `EmailMessage`s for particular properties, so you should take the time to read it over before proceeding. You will not need to use all of the `EmailMessage`'s member functions in your solution.

`nstream` is a custom stream class that allows you to send and receive data over a network connection. It is a stream class, meaning that all of the stream operations you're familiar with apply to `nstream`. You can read and write data using the stream insertion and extraction operators, test for failure with `.fail()`, read lines of text with `getline`, etc.

Read over the header files for these objects before proceeding to the next task. You'll have a much easier time with the next step if you have a better sense of what objects you'll be working with.

Step 2: Implement `UserName` and `SendMail`

Once you've familiarized yourself with the relevant parts of the codebase, you can begin working on the final portion of the assignment: actually sending email messages. You will need to implement two functions from the `mail.h` header. The first, `UserName`, should return your email address; this function will almost certainly be a single line of code. The second, `SendMail`, negotiates an SMTP session and transmits an email message. You will probably want to refer to the earlier section on the structure of SMTP sessions, along with the `nstream.h` and `EmailMessage.h` headers for information about the classes you will be working with.

If you've reached this point, you should be able to test your implementation by sending actual emails. During testing, please only send emails to yourself – you don't want to bombard your friends' email accounts with unnecessary test emails!

Advice, Tips, and Tricks

Here are a few specific pointers that might make your life a lot easier as you go through this assignment:

- *Don't hesitate to ask questions!* **The point of this assignment is to give you a chance to play around with C++ and build cool software, not to punish you for not understanding a particular library or language detail**. If you're having trouble understanding the starter code, run into inexplicable runtime errors, or can't seem to get some part of the assignment working, please send me an email or talk to me after lecture. I genuinely love this material and want to help you learn it, so don't hesitate to ask questions if you need to.

- *Debugging this program may be more difficult than usual*. Communications sent over an `nstream` through a network are not logged anywhere, and so you will not be able to see what messages your program sends and receive. Also, because the program is a GUI and not a console program, you cannot use `cout` to print diagnostic messages as the program runs. To get a better feel for what your program is doing, I recommend logging all messages that your program receives from the server to a special `ofstream` that writes to a log file. If your program doesn't seem to be working correctly, you can then check the log file to get a better view of what your program was doing.

- *Make sure you understand the difference between >> and `getline`.* Remember that the stream extraction operator >> will split text at whitespace boundaries. This means that if you read data from an `nstream` using the stream extraction operator, you will not read in the full text sent by the server. This can cause problems later on. `getline`, on the other hand, will read a full line of text from a stream. Because SMTP servers always reply with complete lines of text, I strongly suggest that you use `getline` to read responses from the server, then use other techniques to extract the reply code.

- *Understand that `nstream` is blocking.* The `nstream` class we've provided you is called a *blocking stream*. If you read data out of an `nstream`, the stream will wait for data to become available from the server to read before returning. Consequently, if you try to read data from the mail server before the mail server has sent any data, you will cause a *deadlock* – the server will be waiting for your program to send it data at the same time that you're waiting for the server to send you data. This means that no data will be sent over the network, and your program will appear to lock up. If your program hangs for fifteen or more seconds when sending data, you may want to verify that you're only reading data from the server at the proper time.

- *Test on campus.* The Stanford SMTP server is configured to reject all requests that do not originate from computers on the Stanford network. If you are off-campus, you will not be able to connect to smtp.stanford.edu and will have to use another mail server. Contact me if this is the case and I can try to set you up with a different mail server.

Deliverables

To submit the assignment, email your `mail.cpp` file, along with any other source files you've modified, to htiek@cs.stanford.edu. If you've added any extensions or special features I should be aware of, let me know in the body of your email. Feel free to email in your source files using your implementation of SendMail; what a fitting use for your program!

Good luck!