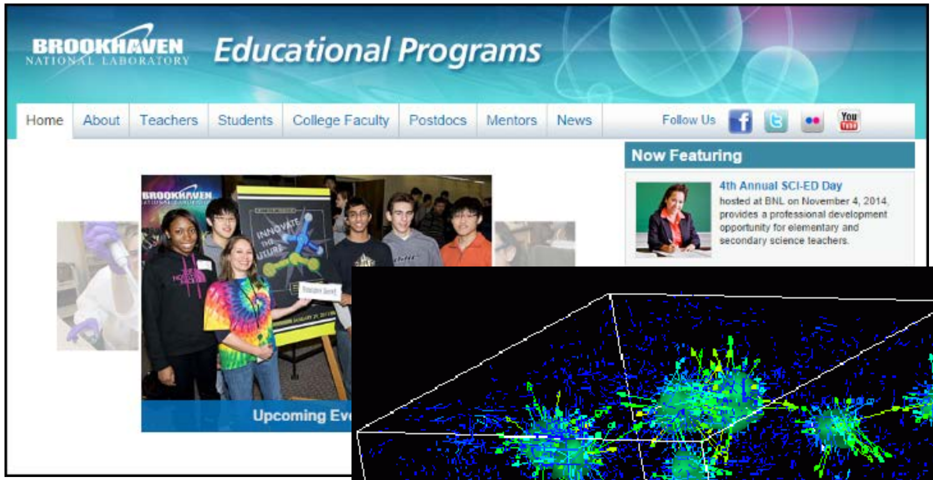
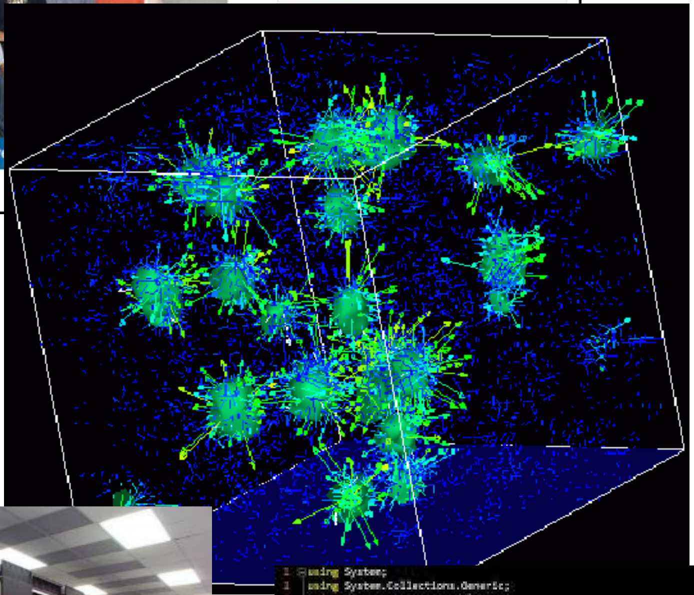
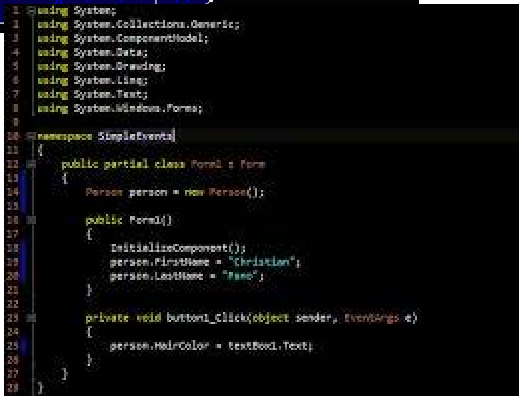


4th Annual SCI-ED Day
hosted at BNL on November 4, 2014, provides a professional development opportunity for elementary and secondary science teachers.





```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleEvents
{
    public partial class Form1 : Form
    {
        Person person = new Person();

        public Form1()
        {
            InitializeComponent();
            person.FirstName = "Christian";
            person.LastName = "Nano";
        }

        private void button1_Click(object sender, EventArgs e)
        {
            person.HairColor = textBox1.Text;
        }
    }
}
```

# Survey of Scientific Computing
(SciComp 301)

Dave Biersach
Brookhaven National Laboratory
dbiersach@bnl.gov

**Session 14**
Cryptanalysis, Anagrams

## Session Goals

- Manage a C++ **string** as a **vector<char>**
  - Understand **ASCII** as an encoding mechanism
  - Read an ASCII **text file** stored on disk into a memory buffer
  - Generate a **histogram of character frequencies** within a file
- Encrypt and decrypt files using "**Caesar Shift**"
  - Perform **bigram analysis** on unreadable cipher text to determine the author's native language
- Generate and discover simple and compound **anagrams**
  - Avoid combinatorial (exponential) explosion in search space

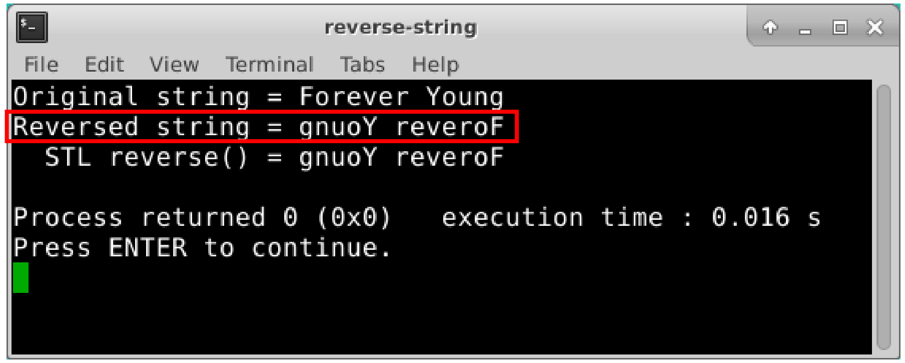
# C++ strings

- A C++ string is mostly equivalent to a **vector<char>**
  - A C++ **char** data type holds one “character”
  - There is a difference between the length of a string (the number of characters in the string) and the number of bytes required to store it in memory or on disk
  - The memory size (number of bits) of a character can vary by platform (Windows vs. Linux)
- A string has **.size()** or **.length()** methods to get the number of *characters* in the string
- We can access individual characters using the **.at()** method

# Reverse a String

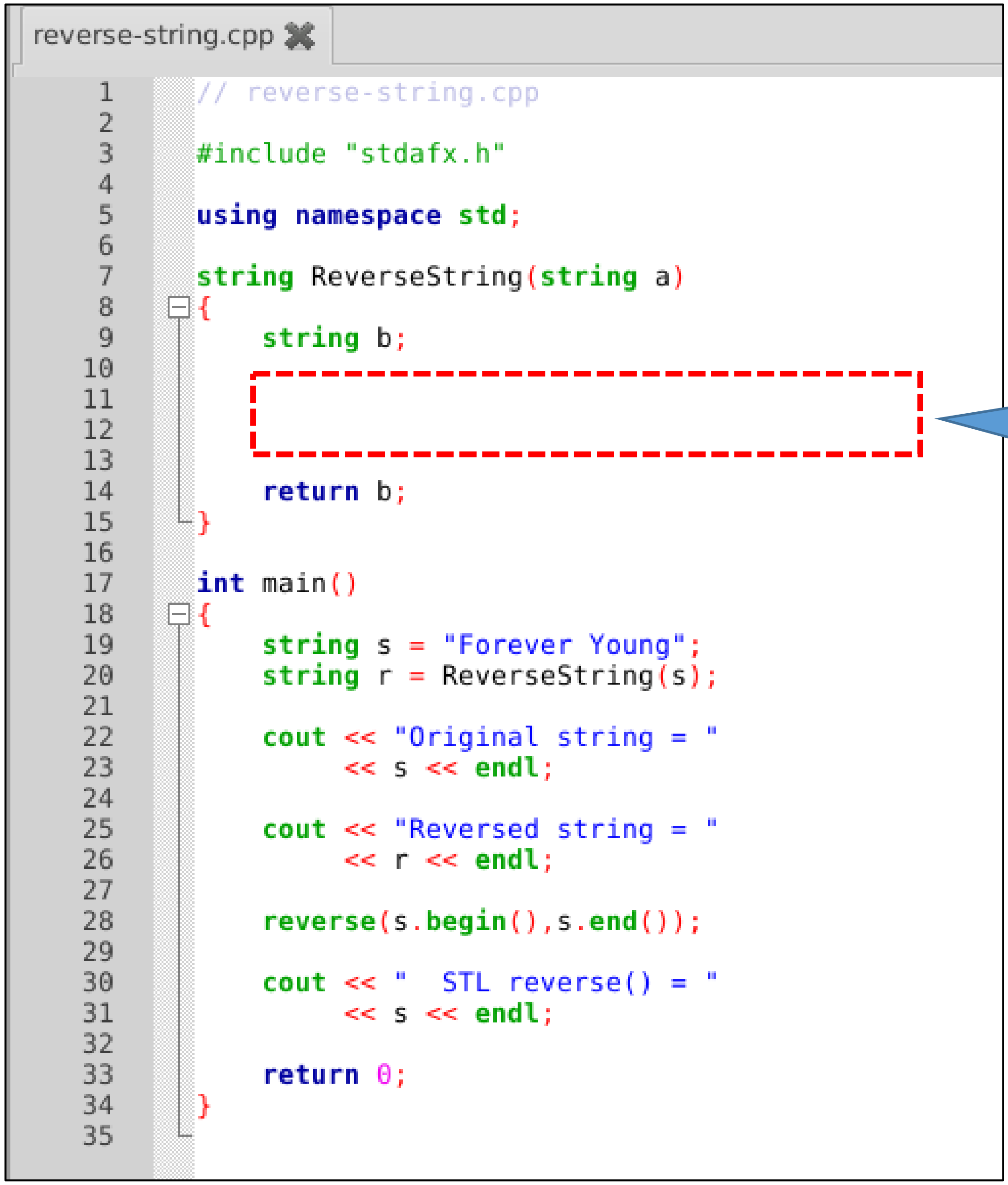
| i | s.at() |
|---|---|
| 0 | F |
| 1 | o |
| 2 | r |
| 3 | e |
| 4 | v |
| 5 | e |
| 6 | r |
| 7 | *(space)* |
| 8 | Y |
| 9 | o |
| 10 | u |
| 11 | n |
| 12 | g |

**s.length()==13**



```
Original string = Forever Young
Reversed string = gnuoY reveroF
  STL reverse() = gnuoY reveroF

Process returned 0 (0x0)   execution time : 0.016 s
Press ENTER to continue.
```

# Open Lab 1 Reverse String



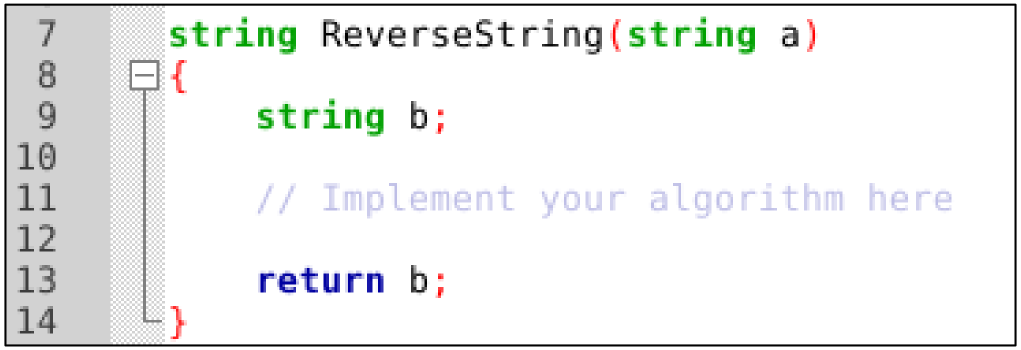
reverse-string.cpp

```
// reverse-string.cpp

#include "stdafx.h"

using namespace std;

string ReverseString(string a)
{
    string b;




    return b;
}

int main()
{
    string s = "Forever Young";
    string r = ReverseString(s);

    cout << "Original string = "
         << s << endl;

    cout << "Reversed string = "
         << r << endl;

    reverse(s.begin(),s.end());

    cout << "  STL reverse() = "
         << s << endl;

    return 0;
}
```

Walk backwards through the given string **a**, while dynamically building string **b** one character at a time

# Edit Lab 1 – Reverse String

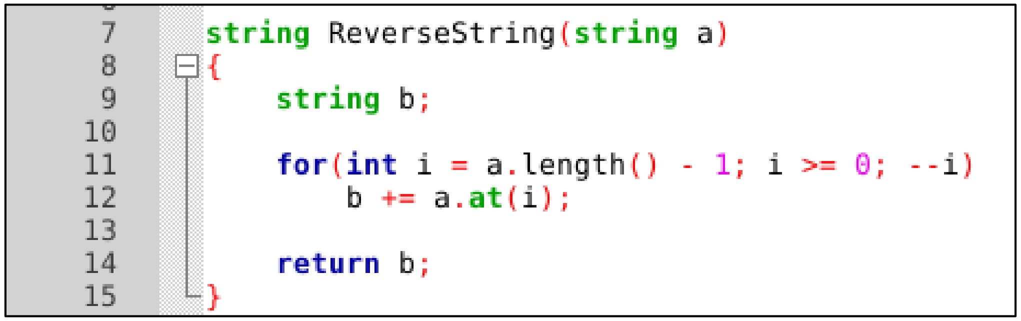
- Add code only to the **ReverseString()** function – don't modify any code in main()
  - A **string** is an vector – you can access individual elements (characters) using the **.at()** method
  - Use **.length()** to get the number of characters in the string
  - You can concatenate strings with the **+** operator
- On return, b should be the string **a** in reverse char order



```
string ReverseString(string a)
{
    string b;

    // Implement your algorithm here

    return b;
}
```

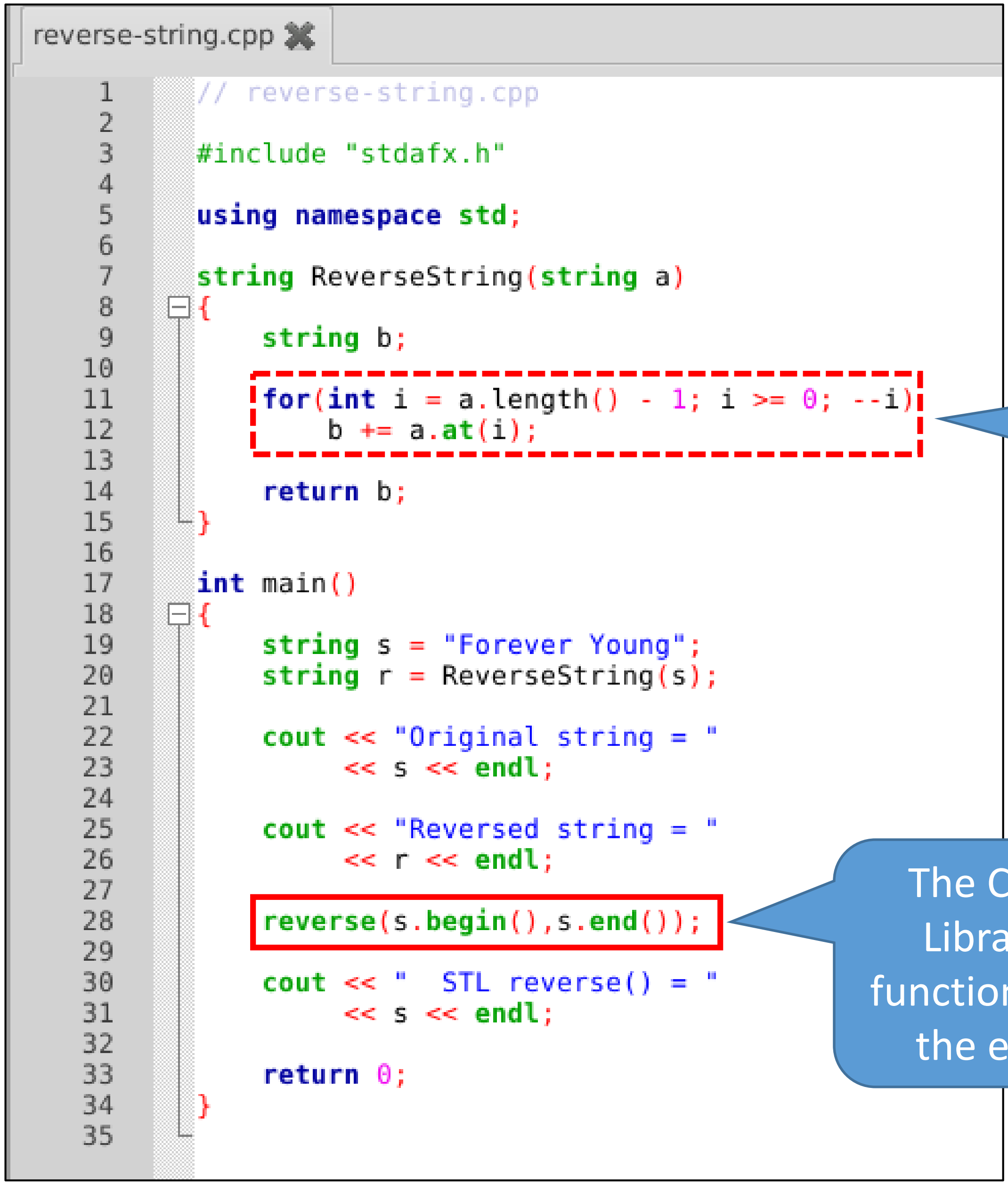
# Edit Lab 1 – Reverse String

- Add code only to the **ReverseString()** function – don't modify any code in main()
  - A **string** is an vector – you can access individual elements (characters) using the **.at()** method
  - Use **.length()** to get the number of characters in the string
  - You can concatenate strings with the **+** operator
- On return, b should be the string a in reverse char order



```
string ReverseString(string a)
{
    string b;

    for(int i = a.length() - 1; i >= 0; --i)
        b += a.at(i);

    return b;
}
```

# Edit Lab 1 Reverse String



reverse-string.cpp

```
// reverse-string.cpp

#include "stdafx.h"

using namespace std;

string ReverseString(string a)
{
    string b;

    for(int i = a.length() - 1; i >= 0; --i)
        b += a.at(i);

    return b;
}

int main()
{
    string s = "Forever Young";
    string r = ReverseString(s);

    cout << "Original string = "
         << s << endl;

    cout << "Reversed string = "
         << r << endl;

    reverse(s.begin(),s.end());

    cout << "  STL reverse() = "
         << s << endl;

    return 0;
}
```

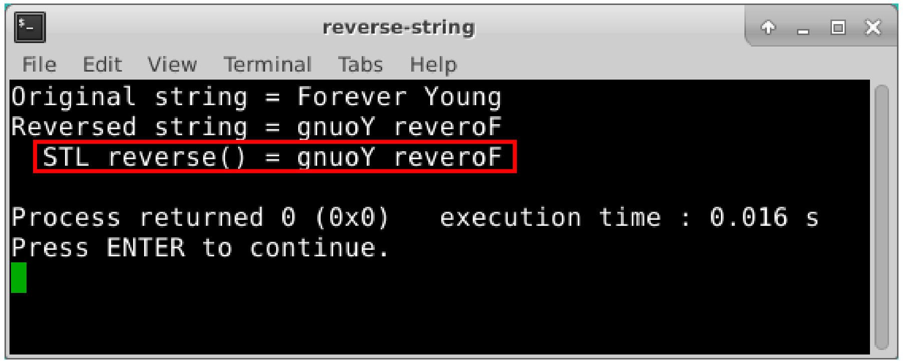
Walk backwards through the given string a, while dynamically building string b one character at a time

The C++ Standard Template Library **(STL)** has a built-in function to reverse the order of the elements in any vector

## Run Lab 1 – Reverse String

| i | s.at() |
|---|---|
| 0 | F |
| 1 | o |
| 2 | r |
| 3 | e |
| 4 | v |
| 5 | e |
| 6 | r |
| 7 | *(space)* |
| 8 | Y |
| 9 | o |
| 10 | u |
| 11 | n |
| 12 | g |

**s.length()==13**



```
Original string = Forever Young
Reversed string = gnuoY reveroF
  STL reverse() = gnuoY reveroF

Process returned 0 (0x0)   execution time : 0.016 s
Press ENTER to continue.
```

**STL** = **S**tandard **T**emplate **L**ibrary

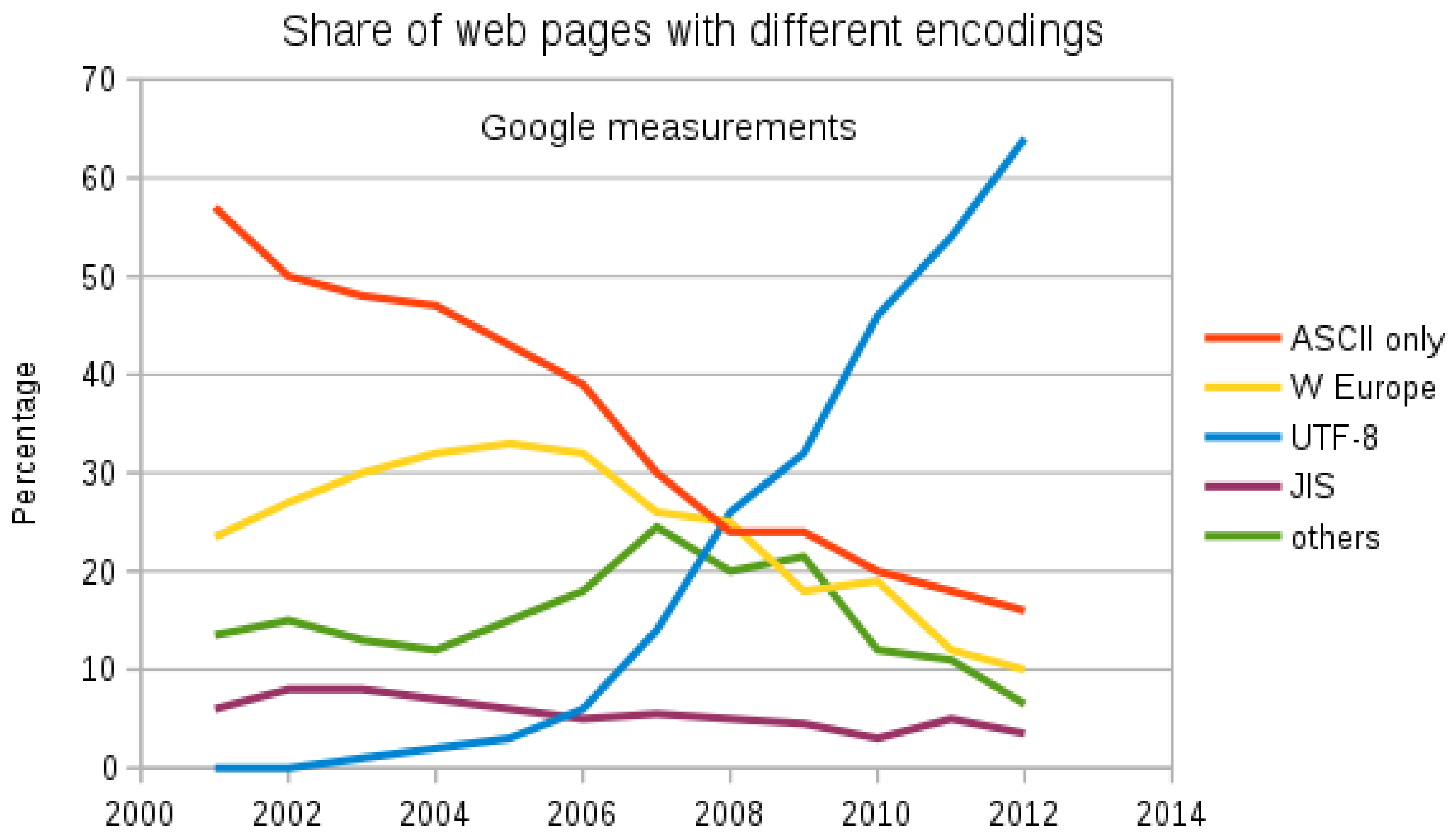
The STL is a set of free & open-source functions and classes to ***reduce*** the amount of code C++ programmers must write to solve common problems

# ASCII (“as-key”)

- **A**merican **S**tandard **C**ode for **I**nformation **I**nterchange
  - ASCII was the most common legacy International standard used across the Internet until 2007
  - Since 2008 ASCII has been surpassed by **UTF-8** (Universal Transformation Format), which includes ASCII as a subset
- ASCII is an 8-bit (**one byte**) character encoding scheme
  - ASCII maps most of the characters in the (Western) languages descending from Latin to a specific integer value
  - There is a 1:1 correspondence between a letter and a number. In ASCII, every character is **always** one byte long (in UTF-8 it is variable)
  - Inside a computer, **all** letters, punctuation marks, even numbers (when treated as strings) are encoded as either ASCII or UTF

# Learn about **UTF-8**

https://en.wikipedia.org/wiki/UTF-8

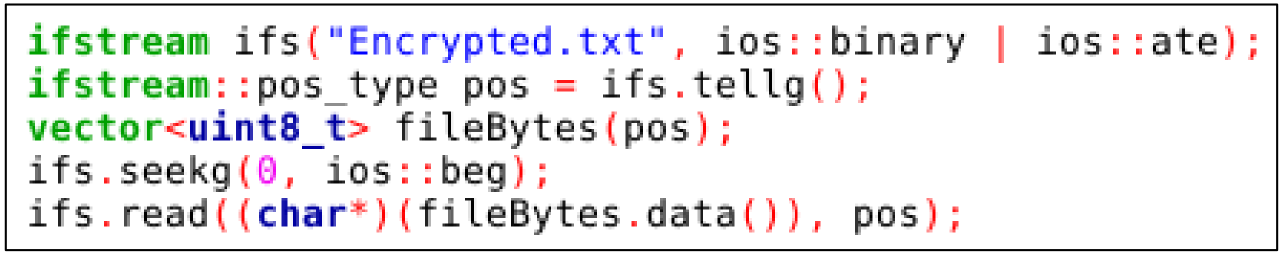


## ASCII range for common English characters

| Dec | Char | Dec | Char | Dec | Char | Dec | Char | Dec | Char |
|---|---|---|---|---|---|---|---|---|---|
| 32 | (space) | 51 | 3 | 70 | F | 89 | Y | 108 | l |
| 33 | ! | 52 | 4 | 71 | G | 90 | Z | 109 | m |
| 34 | " | 53 | 5 | 72 | H | 91 | [ | 110 | n |
| 35 | # | 54 | 6 | 73 | I | 92 | \ | 111 | o |
| 36 | $ | 55 | 7 | 74 | J | 93 | ] | 112 | p |
| 37 | % | 56 | 8 | 75 | K | 94 | ^ | 113 | q |
| 38 | & | 57 | 9 | 76 | L | 95 | _ | 114 | r |
| 39 | ' | 58 | : | 77 | M | 96 | ` | 115 | s |
| 40 | ( | 59 | ; | 78 | N | 97 | a | 116 | t |
| 41 | ) | 60 | < | 79 | O | 98 | b | 117 | u |
| 42 | * | 61 | = | 80 | P | 99 | c | 118 | v |
| 43 | + | 62 | > | 81 | Q | 100 | d | 119 | w |
| 44 | , | 63 | ? | 82 | R | 101 | e | 120 | x |
| 45 | - | 64 | @ | 83 | S | 102 | f | 121 | y |
| 46 | . | 65 | A | 84 | T | 103 | g | 122 | z |
| 47 | / | 66 | B | 85 | U | 104 | h | 123 | { |
| 48 | 0 | 67 | C | 86 | V | 105 | i | 124 | \| |
| 49 | 1 | 68 | D | 87 | W | 106 | j | 125 | } |
| 50 | 2 | 69 | E | 88 | X | 107 | k | 126 | ~ |

# ASCII Text Files – A "stream"

- A file on disk is essentially just a big `byte` **array**
  - A `byte` is an 8-bit *unsigned integer* between 0 and 255 (`uint8_t`)
  - We can declare a byte array and load it with the contents of a file
  - A **stream** of file bytes in memory is called a **buffer**



```
ifstream ifs("Encrypted.txt", ios::binary | ios::ate);
ifstream::pos_type pos = ifs.tellg();
vector<uint8_t> fileBytes(pos);
ifs.seekg(0, ios::beg);
ifs.read((char*)(fileBytes.data()), pos);
```

- We can then access any individual character within the disk file by using the normal **.at() method** on the vector **fileBytes** and specifying an **index value**

## Creating a Frequency Histogram

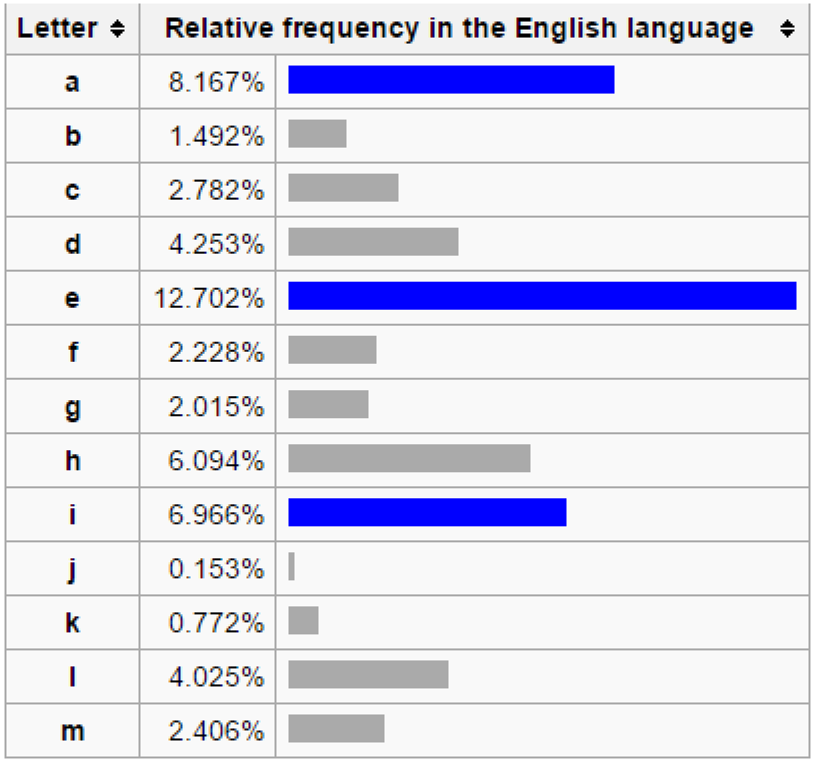
- Consider Lincoln's Gettysburg Address:

  Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

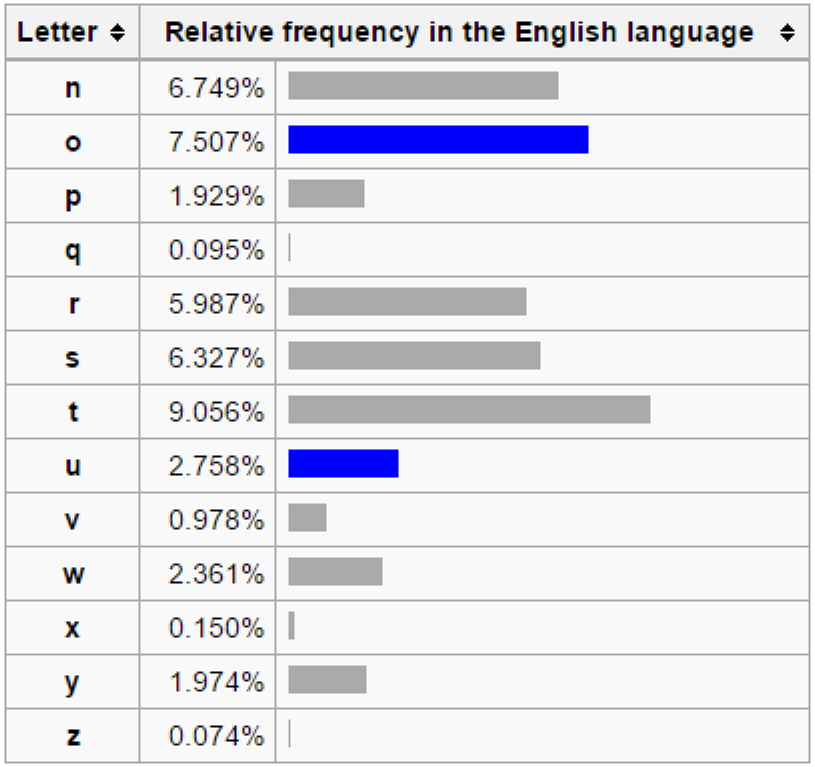
  Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure…

- We want to perform a **histogram analysis** of Lincoln's speech
  - What **letter** do you think occurs most frequently in **English**?
  - Spaces (**ASCII value 32**) usually occur most often because we use spaces as a word breaker

## Letter Frequencies in the English Language



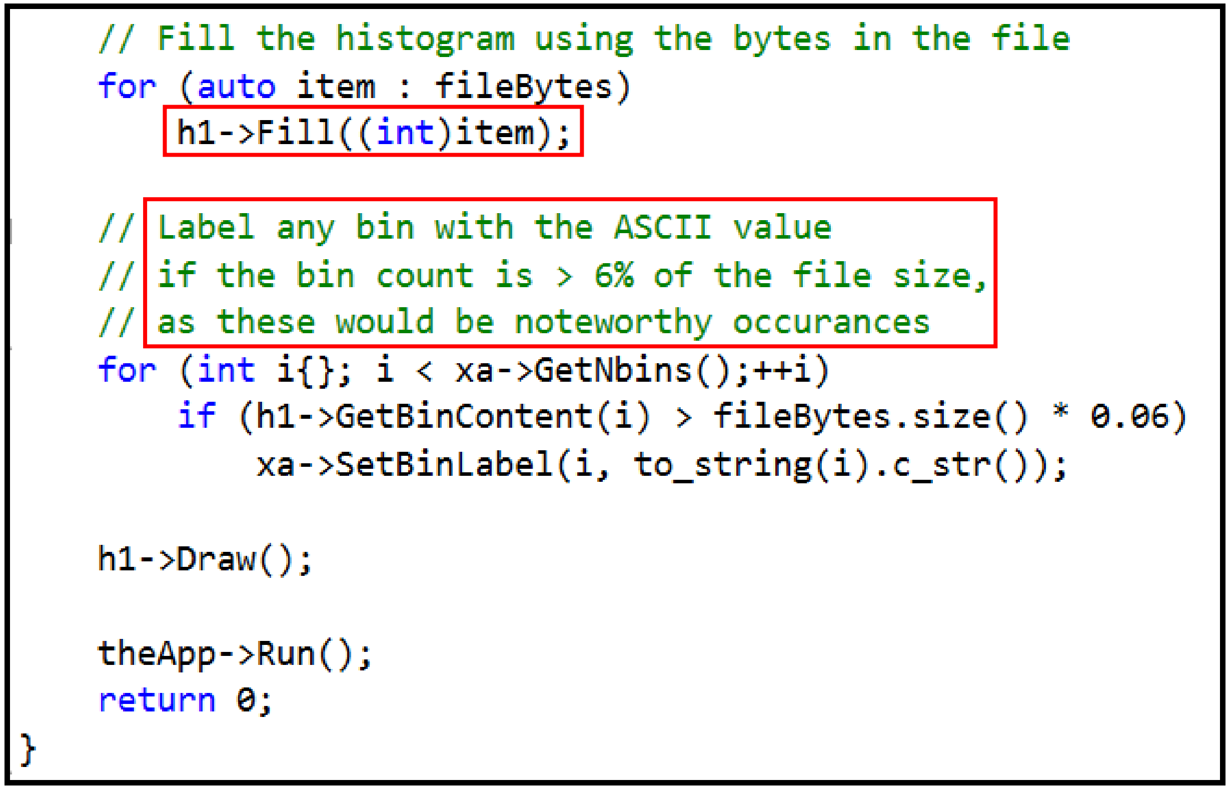
| Letter | Relative frequency in the English language |
|---|---|
| a | 8.167% |
| b | 1.492% |
| c | 2.782% |
| d | 4.253% |
| e | 12.702% |
| f | 2.228% |
| g | 2.015% |
| h | 6.094% |
| i | 6.966% |
| j | 0.153% |
| k | 0.772% |
| l | 4.025% |
| m | 2.406% |



| Letter | Relative frequency in the English language |
|---|---|
| n | 6.749% |
| o | 7.507% |
| p | 1.929% |
| q | 0.095% |
| r | 5.987% |
| s | 6.327% |
| t | 9.056% |
| u | 2.758% |
| v | 0.978% |
| w | 2.361% |
| x | 0.150% |
| y | 1.974% |
| z | 0.074% |

```cpp
int main()
{
    // Open file at the end so the "get" position will be file size
    ifstream ifs("The Gettysburg Address.txt", ios::binary | ios::ate);
    ifstream::pos_type pos = ifs.tellg();

    // Allocate a vector big enough to hold all the file bytes
    vector<uint8_t> fileBytes(pos);

    // Read in the file from the beginning straight into the vector
    ifs.seekg(0, ios::beg);
    ifs.read((char*)(fileBytes.data()), pos);

    // Create a new CERN ROOT app
    string title = "Frequency Analysis";
    TApplication* theApp = new TApplication(title.c_str(), nullptr, nullptr);

    TCanvas* c1 = new TCanvas(title.c_str());
    c1->SetTitle(title.c_str());

    // Create a ROOT one dimensional histogram of integers
    TH1I* h1 = new TH1I(nullptr, title.c_str(), 256, 0, 257);
    h1->SetStats(kFALSE);

    TAxis* ya = h1->GetYaxis();
    ya->SetTitle("Count");
    ya->CenterTitle();

    TAxis* xa = h1->GetXaxis();
    xa->SetTitle("ASCII Value");
    xa->CenterTitle();
    xa->SetTickSize(0);
```

## Open Lab 2 Frequency Histogram

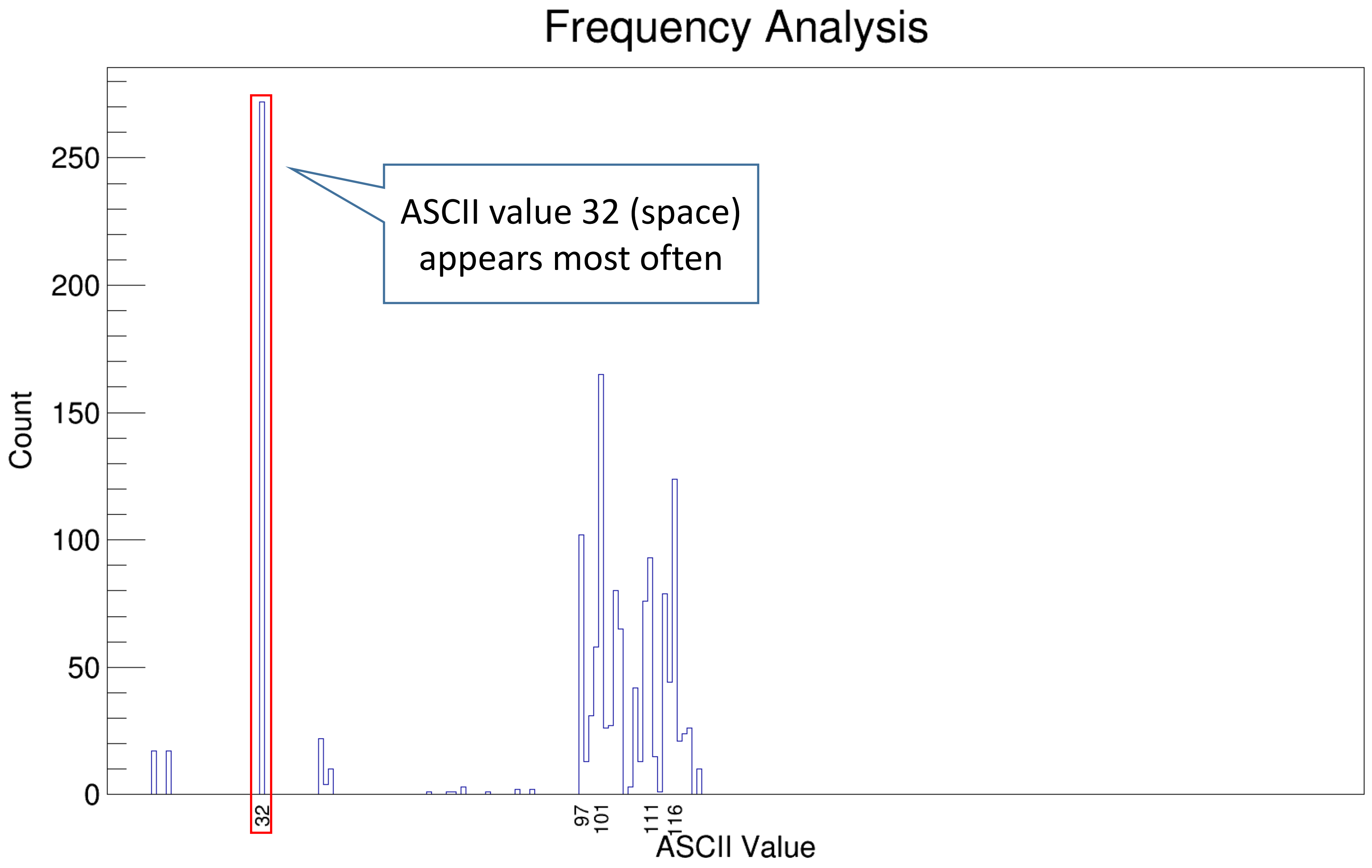


```
    // Fill the histogram using the bytes in the file
    for (auto item : fileBytes)
        h1->Fill((int)item);

    // Label any bin with the ASCII value
    // if the bin count is > 6% of the file size,
    // as these would be noteworthy occurances
    for (int i{}; i < xa->GetNbins();++i)
        if (h1->GetBinContent(i) > fileBytes.size() * 0.06)
            xa->SetBinLabel(i, to_string(i).c_str());

    h1->Draw();

    theApp->Run();
    return 0;
}
```

## **View** Lab 2 Frequency Histogram

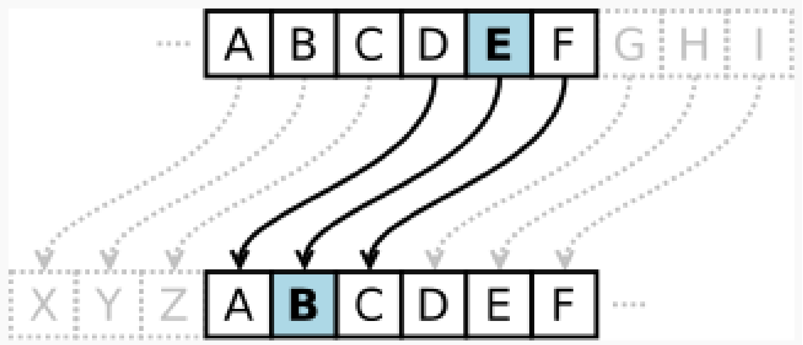
**Run Lab 2**

## Histogram of Lincoln’s Gettysburg Address



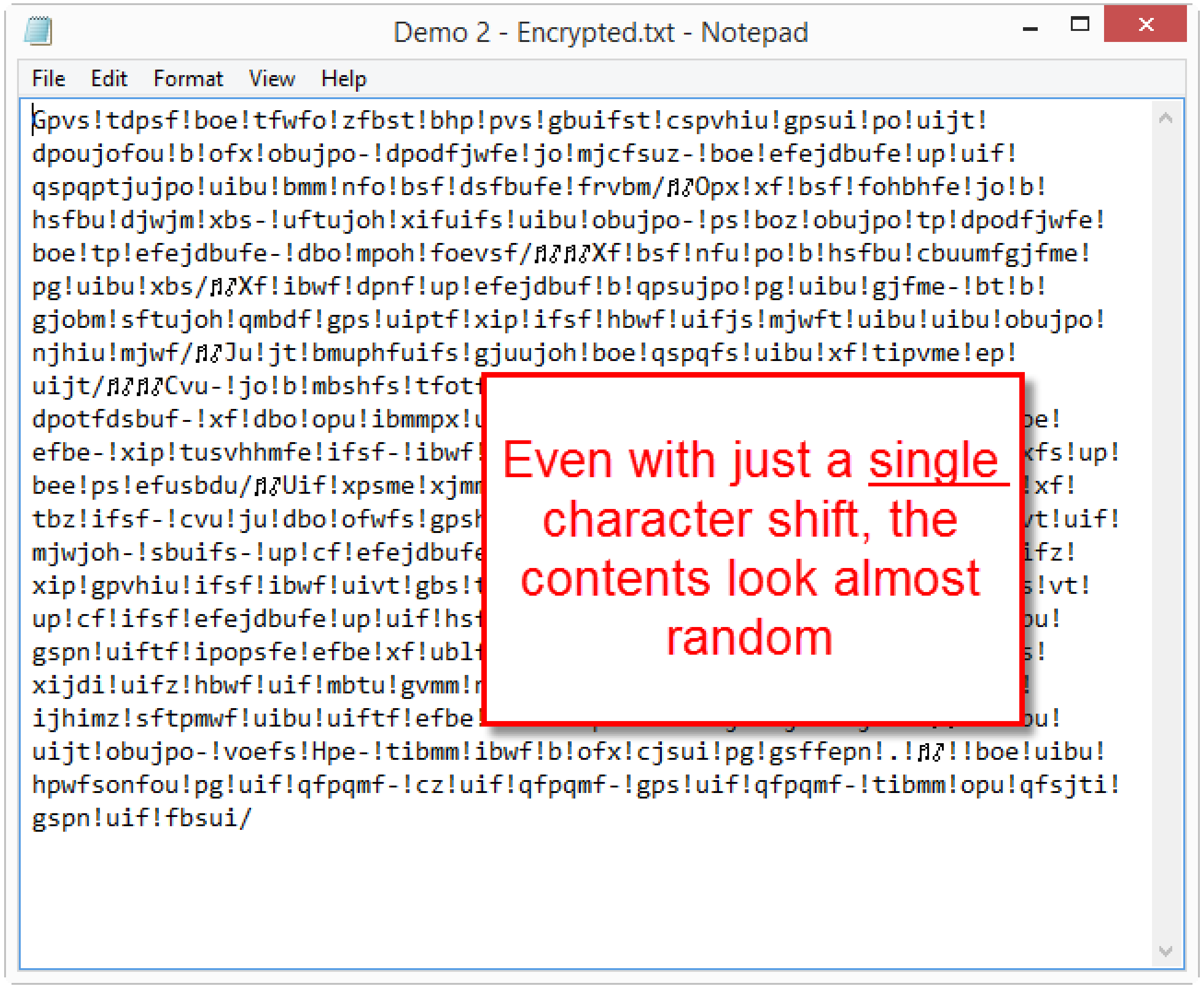
## The Caesar Shift Cipher

- Roman Emperor **Julius Caesar** used a simple (but effective for its time) encryption scheme for his private correspondence
- To create "**cipher text**" from "**plain text**" simply shift the original letters **forward** (***or backward***) a given number of letters in the alphabet
- To decrypt the message, simply **reverse the sign of the shift**



The action of a Caesar cipher is to replace each plaintext letter with a different one a fixed number of places down the alphabet. The cipher illustrated here uses a left shift of three, so that (for example) each occurrence of E in the plaintext becomes B in the ciphertext.

# The Caesar Shift Cipher



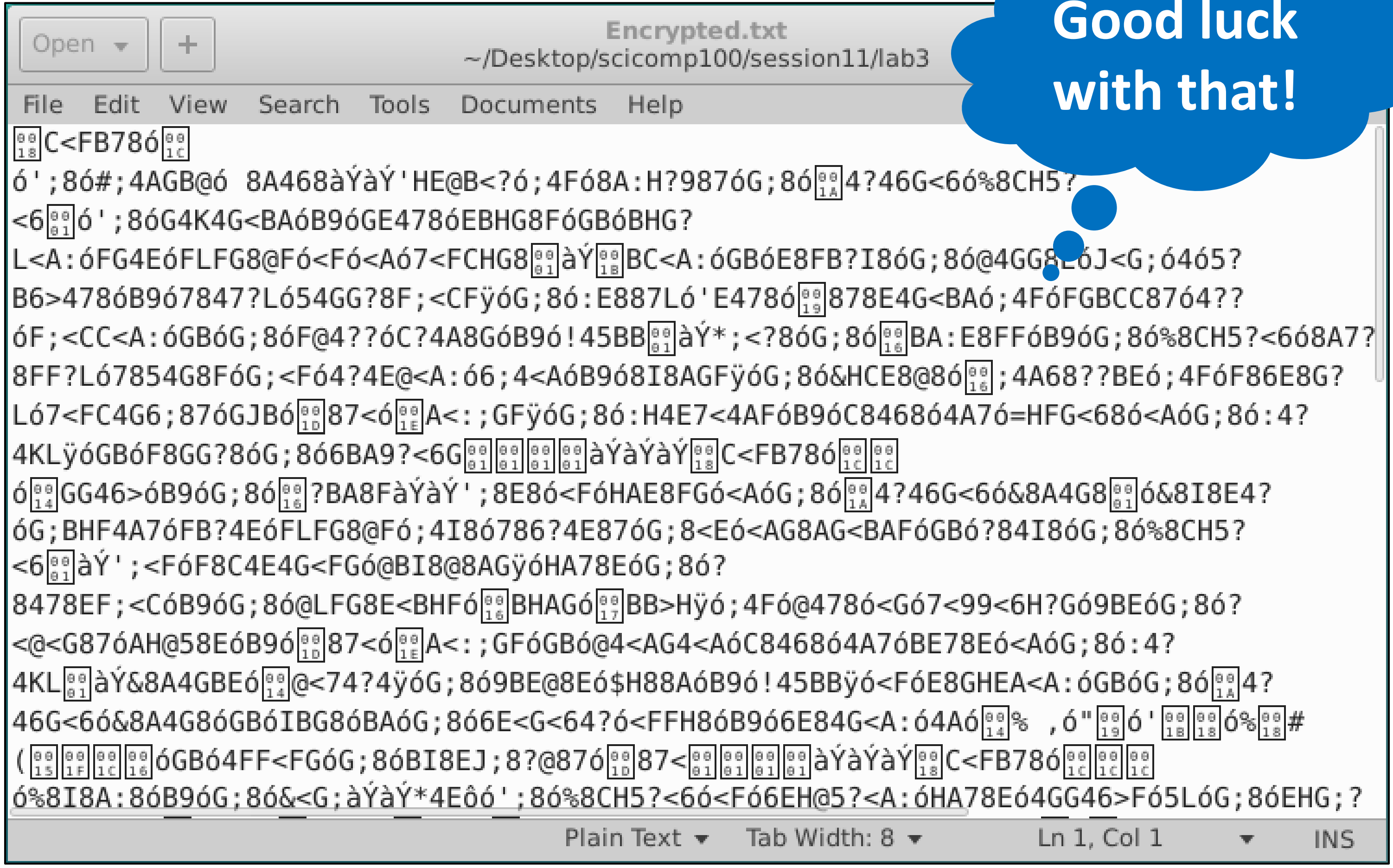
```
Gpvs!tdpsf!boe!tfwfo!zfbst!bhp!pvs!gbuifst!cspvhiu!gpsui!po!uijt!
dpoujofou!b!ofx!obujpo-!dpodfjwfe!jo!mjcfsuz-!boe!efejdbufe!up!uif!
qspqptjujpo!uibu!bmm!nfo!bsf!dsfbufe!frvbm/♫♪Opx!xf!bsf!fohbhfe!jo!b!
hsfbu!djwjm!xbs-!uftujoh!xifuifs!uibu!obujpo-!ps!boz!obujpo!tp!dpodfjwfe!
boe!tp!efejdbufe-!dbo!mpoh!foevsf/♫♪♫♪Xf!bsf!nfu!po!b!hsfbu!cbuumfgjfme!
pg!uibu!xbs/♫♪Xf!ibwf!dpnf!up!efejdbuf!b!qpsujpo!pg!uibu!gjfme-!bt!b!
gjobm!sftujoh!qmbdf!gps!uiptf!xip!ifsf!hbwf!uifjs!mjwft!uibu!uibu!obujpo!
njhiu!mjwf/♫♪Ju!jt!bmuphfuifs!gjuujoh!boe!qspqfs!uibu!xf!tipvme!ep!
uijt/♫♪♫♪Cvu-!jo!b!mbshfs!tfot
dpotfdsbuf-!xf!dbo!opu!ibmmpx!u                                 be!
efbe-!xip!tusvhhmfe!ifsf-!ibwf!                                 xfs!up!
bee!ps!efusbdu/♫♪Uif!xpsme!xjmn                                 !xf!
tbz!ifsf-!cvu!ju!dbo!ofwfs!gpsh                                 vt!uif!
mjwjoh-!sbuifs-!up!cf!efejdbufe                                 ifz!
xip!gpvhiu!ifsf!ibwf!uivt!gbs!t                                 s!vt!
up!cf!ifsf!efejdbufe!up!uif!hst                                 pu!
gspn!uiftf!ipopsfe!efbe!xf!ublf                                 s!
xijdi!uifz!hbwf!uif!mbtu!gvmm!n                                 !
ijhimz!sftpmwf!uibu!uiftf!efbe                                  pu!
uijt!obujpo-!voefs!Hpe-!tibmm!ibwf!b!ofx!cjsui!pg!gsffepn!.!♫♪!!boe!uibu!
hpwfsonfou!pg!uif!qfpqmf-!cz!uif!qfpqmf-!gps!uif!qfpqmf-!tibmm!opu!qfsjti!
gspn!uif!fbsui/
```

Even with just a <u>single</u> character shift, the contents look almost random

# The Caesar Shift Cipher

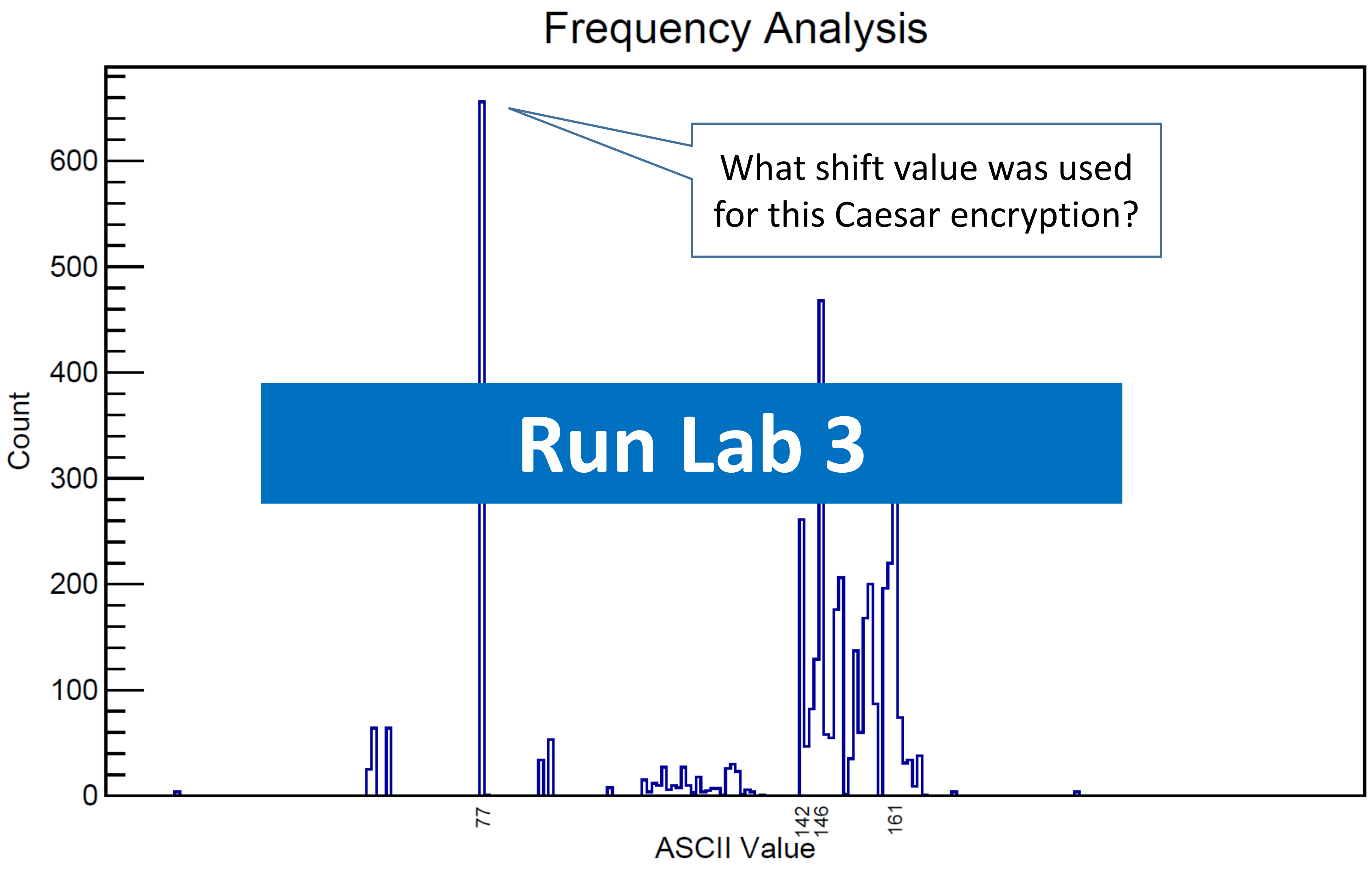
- The question becomes, if we are given a "Caesar Shift" encrypted file, which we believe was written in **English**, how can we figure out **what shift** was used?
- We could use "**brute force**" and try every possible value to see what shift produces legible prose
    - **We only need to try shifts between 1 and 255 – why?**
    - But it would still take a long time to sift through potentially 255 distinct files all filled with *gibberish* in order to break the cipher
- Can we gleam any insight from analyzing the **character histogram** of the <u>encrypted</u> file?

# Open Lab 3 – Ciphertext.txt


Open
+
Encrypted.txt
~/Desktop/scicomp100/session11/lab3
File Edit View Search Tools Documents Help
Plain Text
Tab Width: 8
Ln 1, Col 1
INS
Good luck with that!

```
▯C<FB78ó▯
ó';8ó#;4AGB@ó 8A468àÝàÝ'HE@B<?ó;4Fó8A:H?987óG;8ó▯4?46G<6ó%8CH5?
<6▯ó';8óG4K4G<BAóB9óGE478óEBHG8FóGBóBHG?
L<A:óFG4EóFLFG8@Fó<Fó<Aó7<FCHG8▯àÝ▯BC<A:óGBóE8FB?I8óG;8ó@4GG8EóJ<G;ó4ó5?
B6>478óB9ó7847?Ló54GG?8F;<CFÿóG;8ó:E887Ló'E478ó▯878E4G<BAó;4FóFGBCC87ó4??
óF;<CC<A:óGBóG;8óF@4??óC?4A8GóB9ó!45BB▯àÝ*;<?8óG;8ó▯BA:E8FFóB9óG;8ó%8CH5?<6ó8A7?
8FF?Ló7854G8FóG;<Fó4?4E@<A:ó6;4<AóB9ó8I8AGFÿóG;8ó&HCE8@8ó▯;4A68??BEó;4FóF86E8G?
Ló7<FC4G6;87óGJBó▯87<ó▯A<:;GFÿóG;8ó:H4E7<4AFóB9óC846864A7ó=HFG<68ó<AóG;8ó:4?
4KLÿóGBóF8GG?8óG;8ó6BA9?<6G▯▯▯▯àÝàÝàÝ▯C<FB78ó▯▯
ó▯GG46>óB9óG;8ó▯?BA8FàÝàÝ';8E8ó<FóHAE8FGó<AóG;8ó▯4?46G<6ó&8A4G8▯ó&8I8E4?
óG;BHF4A7óFB?4EóFLFG8@Fó;4I8ó786?4E87óG;8<Eó<AG8AG<BAFóGBó?84I8óG;8ó%8CH5?
<6▯àÝ';<FóF8C4E4G<FGó@BI8@8AGÿóHA78EóG;8ó?
8478EF;<CóB9óG;8ó@LFG8E<BHFó▯BHAGó▯BB>Hÿó;4Fó@478ó<Gó7<99<6H?Góó9BEóG;8ó?
<@<G87óAH@58EóB9ó▯87<ó▯A<:;GFóGBó@4<AG4<AóC846864A7óBE78Eó<AóG;8ó:4?
4KL▯àÝ&8A4GBEó▯@<74?4ÿóG;8ó9BE@8Eó$H88AóB9ó!45BBÿó<FóE8GHEA<A:óGBóG;8ó▯4?
46G<6ó&8A4G8óGBóIBG8óBAóG;8ó6E<G<64?ó<FFH8óB9ó6E84G<A:ó4Aó▯% ,ó"▯ó'▯▯ó%▯#
(▯▯▯▯óGBó4FF<FGóG;8óBI8EJ;8?@87ó▯87<▯▯▯▯àÝàÝàÝ▯C<FB78ó▯▯▯
ó%8I8A:8óB9óG;8ó&<G;àÝàÝ*4Eôó';8ó%8CH5?<6ó<Fó6EH@5?<A:óHA78Eó4GG46>Fó5LóG;8óEHG;?
```

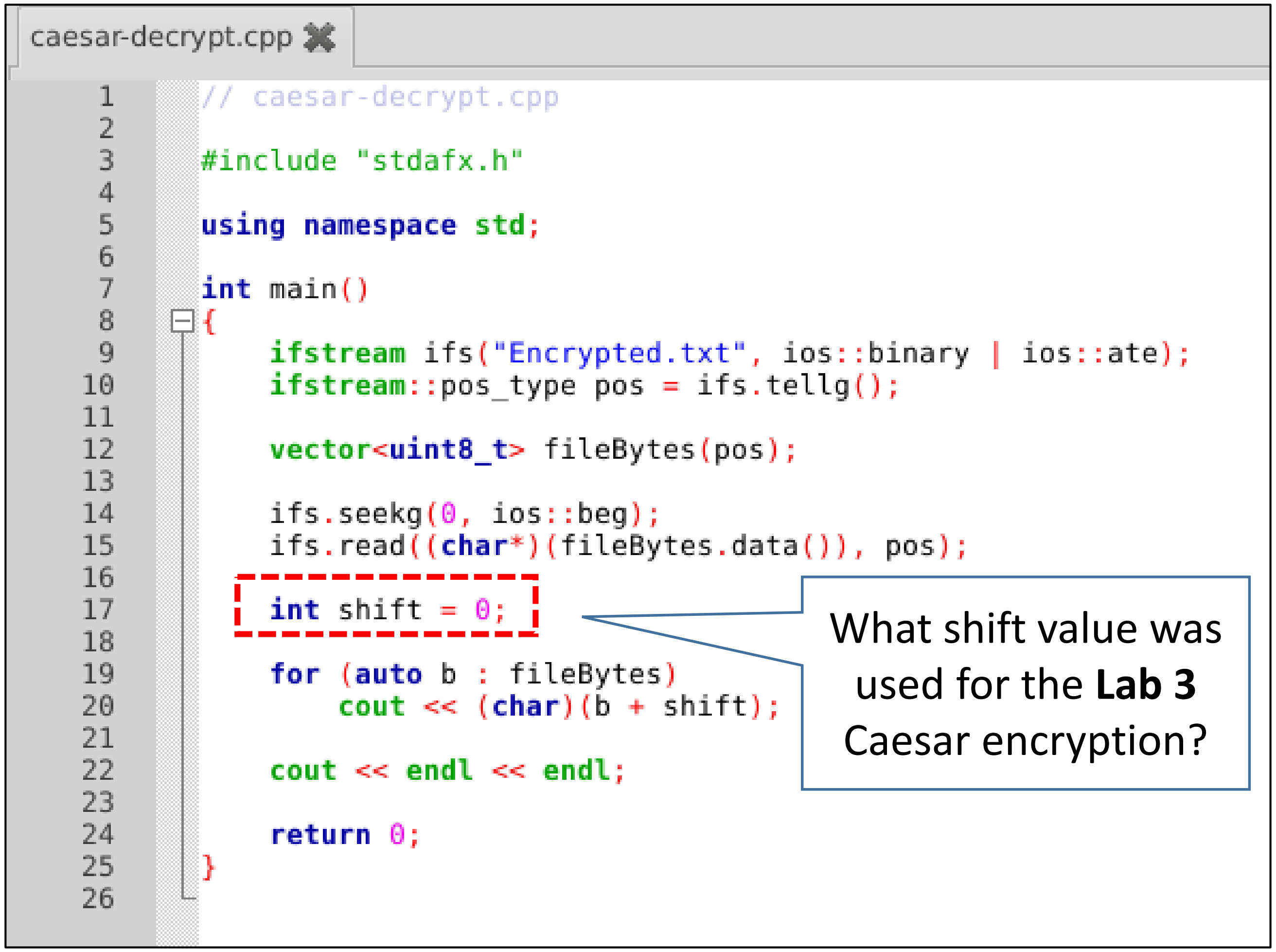
## Run Lab 3 – Histogram of Ciphertext



## **Open** Lab 4 – Caesar Decrypt

- **Your mission is to decrypt the Lab 3 ciphertext file**
- What if the survival of your country depended upon your ability to crack the encryption?

# Edit Lab 4 – Caesar Decrypt

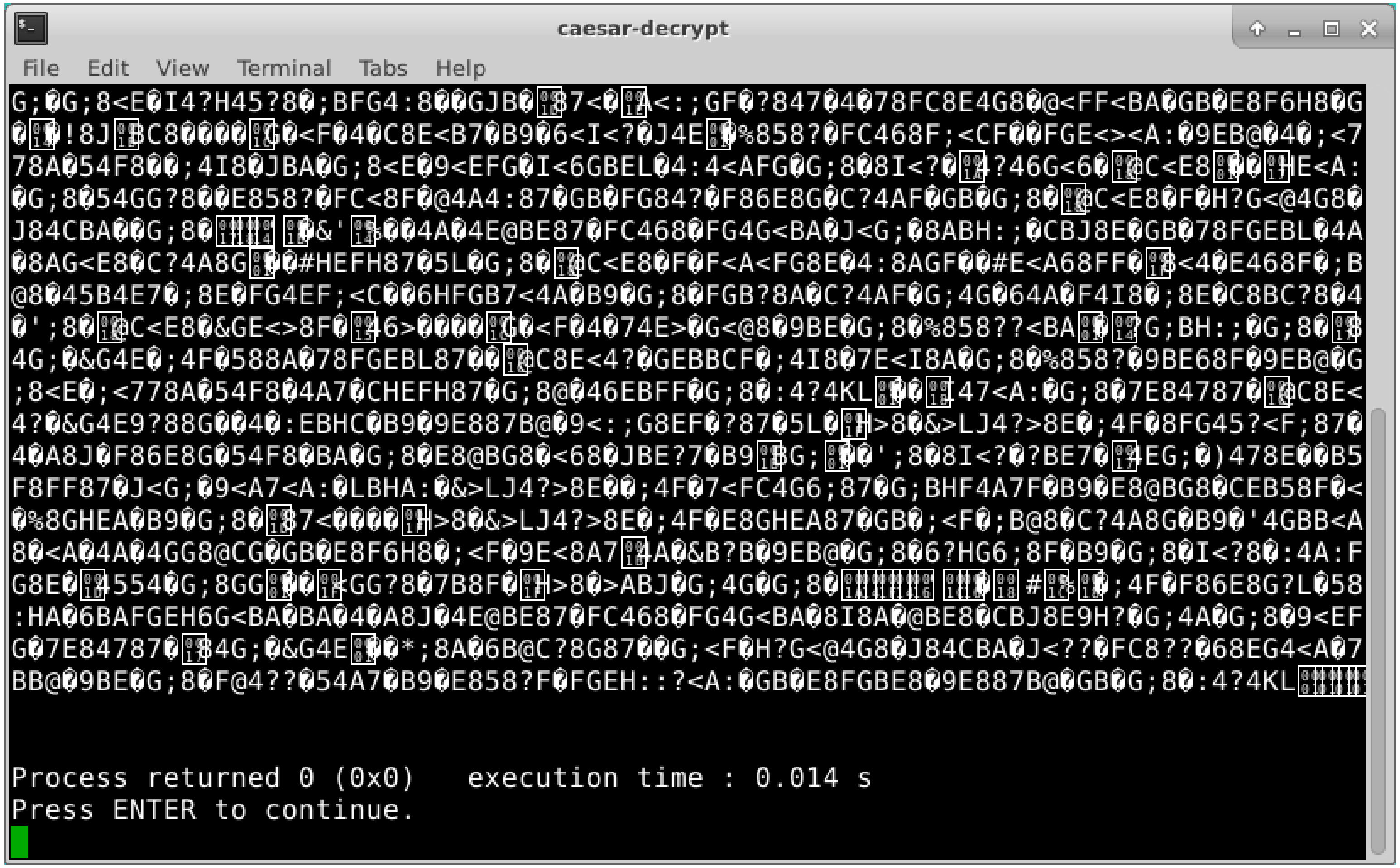


caesar-decrypt.cpp

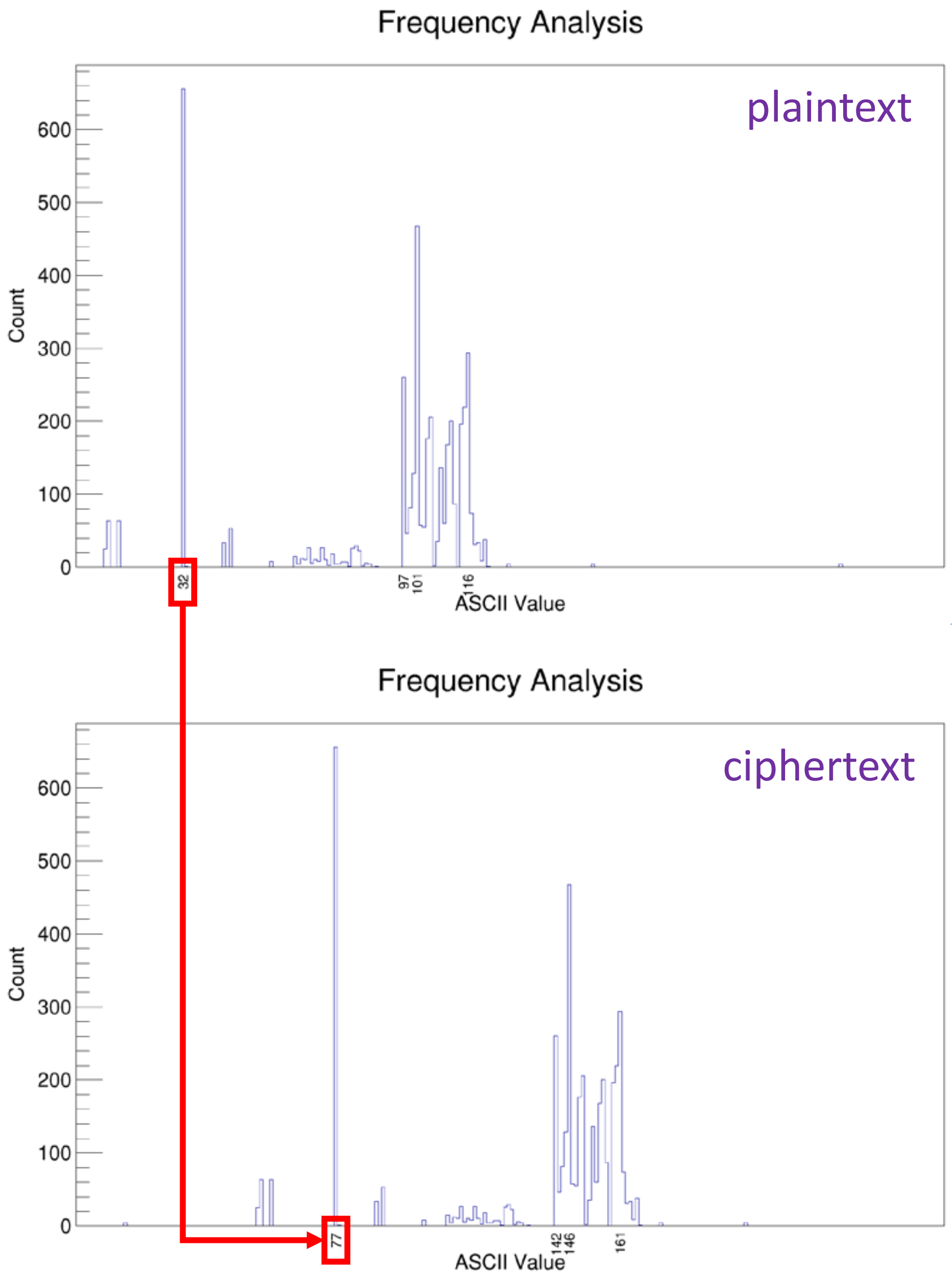
```
// caesar-decrypt.cpp

#include "stdafx.h"

using namespace std;

int main()
{
    ifstream ifs("Encrypted.txt", ios::binary | ios::ate);
    ifstream::pos_type pos = ifs.tellg();

    vector<uint8_t> fileBytes(pos);

    ifs.seekg(0, ios::beg);
    ifs.read((char*)(fileBytes.data()), pos);

    int shift = 0;

    for (auto b : fileBytes)
        cout << (char)(b + shift);

    cout << endl << endl;

    return 0;
}

```

What shift value was used for the **Lab 3** Caesar encryption?

# Run Lab 4 – Caesar Decrypt



```
G;�G;8<E�I4?H45?8�;BFG4:8��GJB�□87<�□A<:;GF�?847�4�78FC8E4G8�@<FF<BA�GB�E8F6H8�G
�□�!8J□BC8�����□G�<F�4�C8E<B7�B9�6<I<?�J4E□�%858?�FC468F;<CF��FGE<><A:�9EB@�4�;<7
78A�54F8��;4I8�JBA�G;8<E�9<EFG�I<6GBEL�4:4<AFG�G;8�8I<?�□4?46G<6�□@C<E8□��□HE<A:
�G;8�54GG?8��E858?�FC<8F�@4A4:87�GB�FG84?�F86E8G�C?4AF�GB�G;8�□@C<E8�F�H?G<@4G8�
J84CBA��G;8�□□□□ □�&'□%��4A�4E@BE87�FC468�FG4G<BA�J<G;�8ABH:;�CBJ8E�GB�78FGEBL�4A
�8AG<E8�C?4A8G□��#HEFH87�5L�G;8�□@C<E8�F�F<A<FG8E�4:8AGF��#E<A68FF�□8<4�E468F�;B
@8�45B4E7�;8E�FG4EF;<C��6HFGB7<4A�B9�G;8�FGB?8A�C?4AF�G;4G�64A�F4I8�;8E�C8BC?8�4
�';8�□@C<E8�&GE<>8F�□46>����□G�<F�4�74E>�G<@8�9BE�G;8�%858??<BA□□G;BH:;�G;8�□8
4G;�&G4E�;4F�588A�78FGEBL87��□@C8E<4?�GEBBCF�;4I8�7E<I8A�G;8�%858?�9BE68F�9EB@�G
;8<E�;<778A�54F8�4A7�CHEFH87�G;8@�46EBFF�G;8�:4?4KL□��□I47<A:�G;8�7E84787�□@C8E<
4?�&G4E9?88G��4�:EBHC�B9�9E887B@�9<:;G8EF�?87�5L�□H>8�&>LJ4?>8E�;4F�8FG45?<F;87�
4�A8J�F86E8G�54F8�BA�G;8�E8@BG8�<68�JBE?7�B9□BG;□��';8�8I<?�?BE7�□4EG;�)478E��B5
F8FF87�J<G;�9<A7<A:�LBHA:�&>LJ4?>8E��;4F�7<FC4G6;87�G;BHF4A7F�B9�E8@BG8�CEB58F�<
�%8GHEA�B9�G;8�□87<����□H>8�&>LJ4?>8E�;4F�E8GHEA87�GB�;<F�;B@8�C?4A8G�B9�'4GBB<A
8�<A�4A�4GG8@CG�GB�E8F6H8�;<F�9E<8A7□4A�&B?B�9EB@�G;8�6?HG6;8F�B9�G;8�I<?8�:4A:F
G8E�□4554�G;8GG□��□<GG?8�7B8F�□H>8�>ABJ�G;4G�G;8�□□□□□□ □□�□ #□%□�;4F�F86E8G?L�58
:HA�6BAFGEH6G<BA�BA�4�A8J�4E@BE87�FC468�FG4G<BA�8I8A�@BE8�CBJ8E9H?�G;4A�G;8�9<EF
G�7E84787�□84G;�&G4E□��*;8A�6B@C?8G87��G;<F�H?G<@4G8�J84CBA�J<??�FC8??�68EG4<A�7
BB@�9BE�G;8�F@4??�54A7�B9�E858?�F�FGEH::?<A:�GB�E8FGBE8�9E887B@�GB�G;8�:4?4KL□□□□


Process returned 0 (0x0)   execution time : 0.014 s
Press ENTER to continue.
```



Notice the ***relative*** frequencies are the same before and after encryption

## The Caesar Shift Cipher

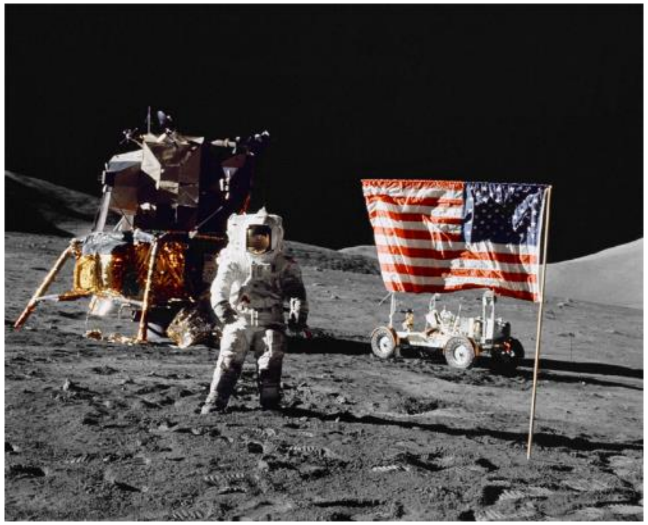
Because the **Caesar Shift** is a **monoalphabetic substitution cipher**, it is susceptible to **cryptanalysis** (breaking) by **frequency analysis**

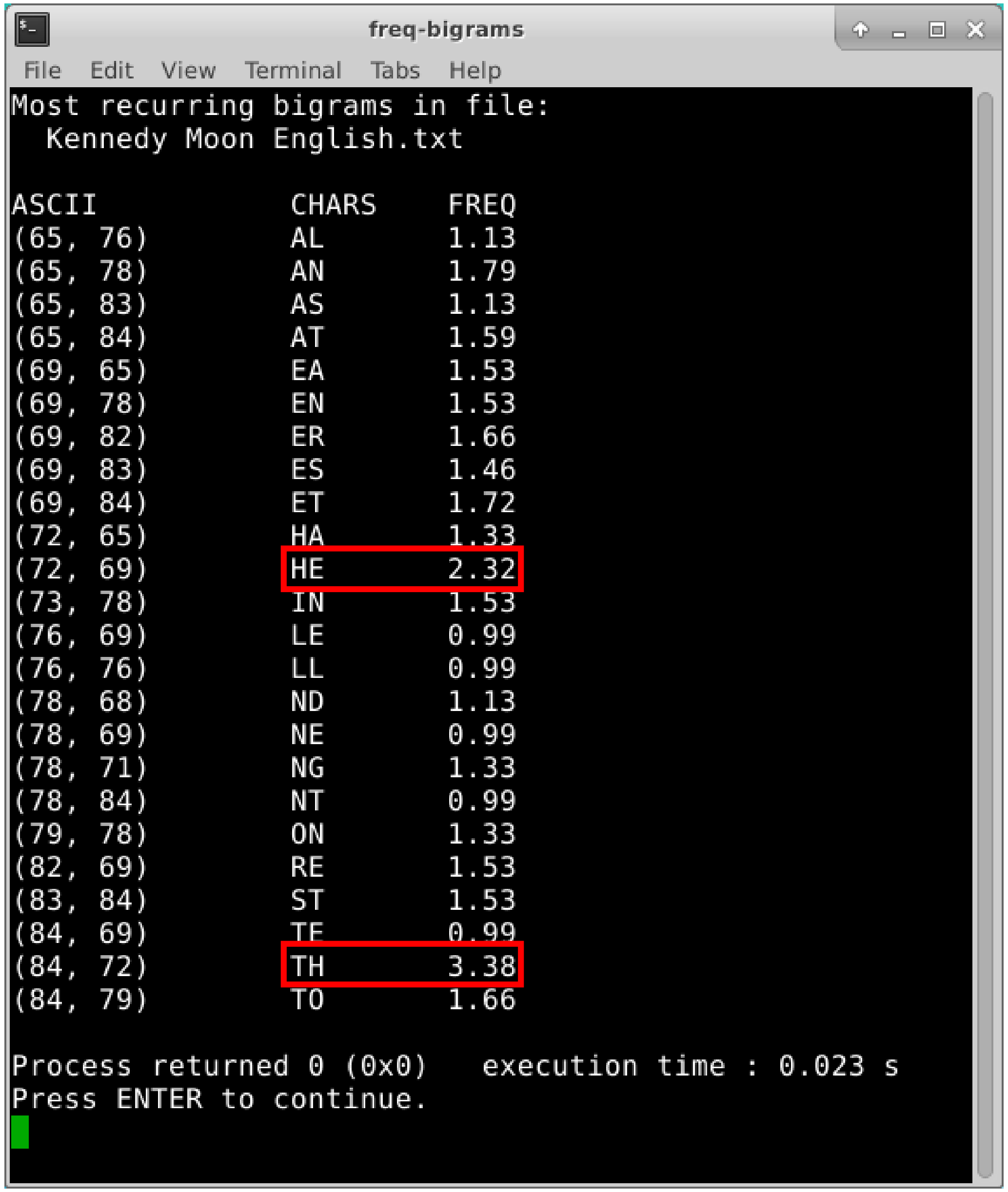
## Bigram Analysis

- Most Western (Latin influenced) languages have a unique fingerprint from their most frequent **bigrams** (two-letter pair)
    - In **English** the bigrams **TH** and **HE** occur most often, since “the” is the most frequent word in English
    - “The” also occurs very often in other languages, though each language spells it differently, and this helps establish the **distinct statistical profiles** of each language
- Linguists and statisticians have compiled lists of the most frequent **bigrams** *per* language
- We will analyze the bigrams in **President Kennedy’s Rice University Speech** - where he set the goal in 1962 for the USA to reach the moon before 1970!

# Kennedy's Moon Speech in 1962

"We choose to go to the moon in this decade and do the other things, not because they are easy, *but because they are hard*, because that goal will serve to organize and measure **the best of our energies and skills**, because that challenge is one that we are willing to accept, one we are unwilling to postpone, and one which we intend to win."





```
Most recurring bigrams in file:
  Kennedy Moon English.txt

ASCII           CHARS    FREQ
(65, 76)        AL       1.13
(65, 78)        AN       1.79
(65, 83)        AS       1.13
(65, 84)        AT       1.59
(69, 65)        EA       1.53
(69, 78)        EN       1.53
(69, 82)        ER       1.66
(69, 83)        ES       1.46
(69, 84)        ET       1.72
(72, 65)        HA       1.33
(72, 69)        HE       2.32
(73, 78)        IN       1.53
(76, 69)        LE       0.99
(76, 76)        LL       0.99
(78, 68)        ND       1.13
(78, 69)        NE       0.99
(78, 71)        NG       1.33
(78, 84)        NT       0.99
(79, 78)        ON       1.33
(82, 69)        RE       1.53
(83, 84)        ST       1.53
(84, 69)        TE       0.99
(84, 72)        TH       3.38
(84, 79)        TO       1.66

Process returned 0 (0x0)   execution time : 0.023 s
Press ENTER to continue.
```

# **Bigram Analysis** of Kennedy's Moon Speech (English)

# Kennedy’s Moon Speech Translated

Nous choisissons d'aller sur la lune. Nous choisissons d'aller sur la lune dans cette décennie et de faire d'autres choses, non pas parce qu'ils sont faciles, mais parce qu'ils sont difficiles, parce que ce but servira à organiser et mesurer le meilleur de nos énergies et de compét
parce que ce défi est l'un
sommes prêts à accepter,
ne sommes pas disposés à
et celui qui nous avons l'i
gagner.

Elegimos ir a la Luna. Elegimos ir a la Luna en esta década y hacer las otras cosas, no porque sean fáciles, sino porque son difíciles, porque esa meta servirá para organizar y medir lo mejor de nuestras energías y habilidades, porque ese desafío es
tos a
dispuestos
hemos la

Wir wählen, um zum Mond zu fliegen. Wir wählen, um zum Mond in diesem Jahrzehnt zu gehen und die anderen Dinge, nicht weil sie leicht sind, sondern weil sie hart sind, denn das Ziel wird dazu dienen, zu organisieren und zu messen, das Beste aus unserer Energien und Fähigkeiten, denn das ist eine Herausforderung dass wir bereit sind zu akzeptieren, das wir nicht bereit sind, zu verschieben, und eine, die wir beabsichtigen, zu gewinnen.

# Bigram Statistics by Language

**Bigrams - Kennedy Speech**

🟨 = Unique Indicators (for each language)
🟩 = Relative Indicator (see German)

| English | | Speech |
|---|---|---|
| TH | 2.71 | 3.38 |
| HE | 2.33 | 2.32 |
| IN | 2.03 | 1.53 |
| ER | 1.78 | 1.66 |
| AN | 1.61 | 1.79 |
| RE | 1.41 | |
| ES | 1.32 | |
| ON | 1.32 | |
| ST | 1.25 | |
| NT | 1.17 | |
| EN | 1.13 | |
| AT | 1.12 | |
| Top 5: | **10.46** | **10.68** |

| Spanish | | Speech |
|---|---|---|
| DE | 2.57 | 2.41 |
| ES | 2.31 | 2.84 |
| EN | 2.27 | 1.75 |
| EL | 2.01 | 1.57 |
| LA | 1.80 | 1.69 |
| OS | 1.79 | |
| ON | 1.61 | |
| AS | 1.56 | |
| ER | 1.52 | |
| RA | 1.47 | |
| AD | 1.43 | |
| AR | 1.43 | |
| Top 5: | **10.96** | **10.26** |

| French | | Speech |
|---|---|---|
| ES | 2.91 | 2.17 |
| LE | 2.08 | 2.17 |
| DE | 2.02 | 2.11 |
| EN | 1.97 | 1.61 |
| ON | 1.70 | 2.00 |
| NT | 1.69 | |
| RE | 1.62 | |
| AN | 1.28 | |
| LA | 1.25 | |
| ER | 1.21 | |
| TE | 1.19 | |
| EL | 1.15 | |
| Top 5: | **10.68** | **10.06** |

| German | | Speech |
|---|---|---|
| ER | 3.90 | 3.29 |
| EN | 3.61 | 4.44 |
| CH | 2.36 | 1.67 |
| DE | 2.31 | 1.90 |
| EI | 1.98 | 1.73 |
| TE | 1.98 | |
| IN | 1.71 | |
| ND | 1.68 | |
| IE | 1.48 | |
| GE | 1.45 | |
| ST | 1.21 | |
| NE | 1.19 | |
| Top 5: | **14.16** | **13.03** |

German is the most consistent language for bigrams

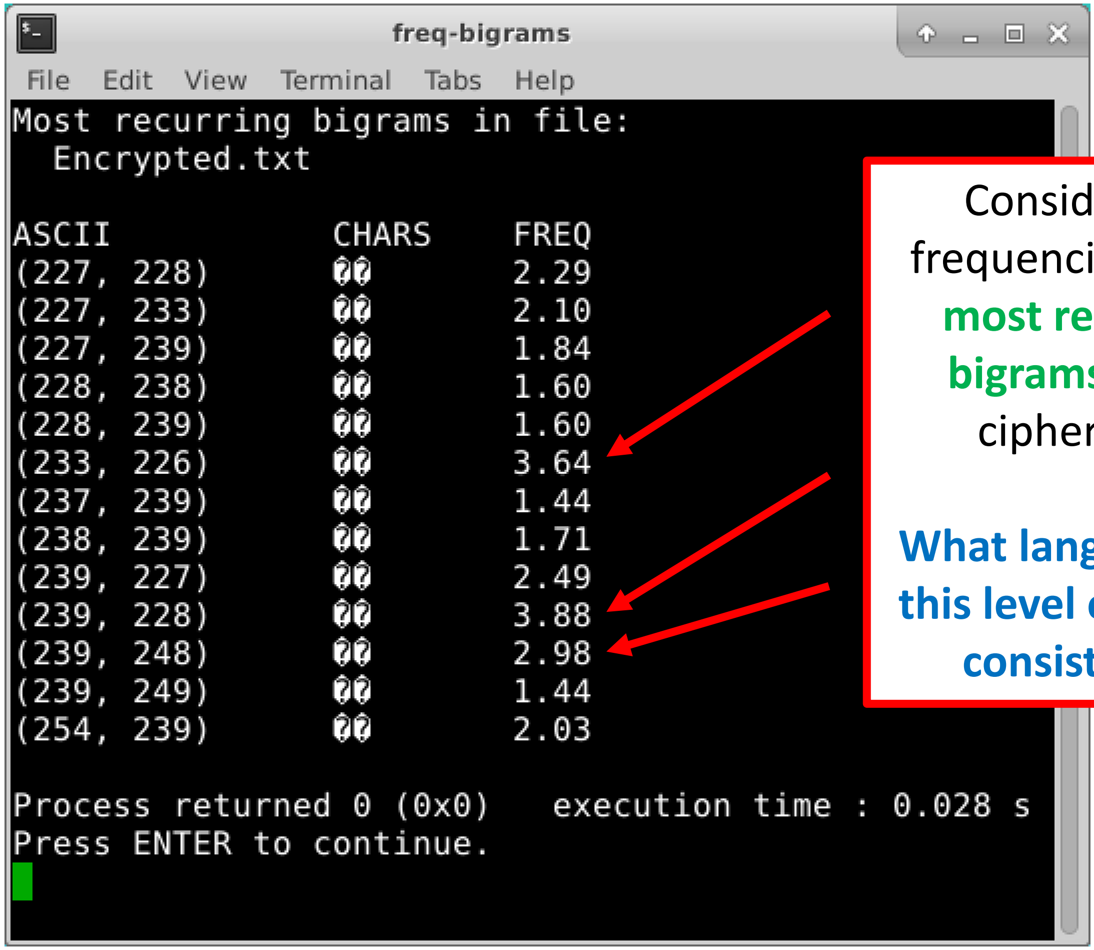
## Bigram Analysis

- You have been given **another** encrypted message!
- It **is** encrypted with a **monoalphabetic substitution cipher**, ***but*** single letter frequency analysis **does not** show any consistent Caesar shifting – it appears to be a ***different shift*** **for each plaintext letter**
- Even if you are ***unable*** to break the encryption, **can you tell what language was used** to author the plaintext?
  - Q: **Are you serious**? How can you possibly discern the author language if you **cannot even read the contents** in the first place?
  - A: Enigma was also **unbreakable**

# Open Lab 5 – Ciphertext.txt

```
áëúãþïæïãäùïøùþïäèæãéáçïãäïçÿþþïøìÿâøçãéâðÿçìæÿíâêìïãçãþîïäìïäùþïø
ùïùúãþðïãéâÿÿøîïëæùëèùéâãïîùíïùþïþøëíïäçïãäïâëäîíïúNéáùþVéáÿëøïãäúé
\ääþïãéâçïãäïäæãïèïäùúøÿäíâëìþâëøïèøëäãïîçÿþþïøìVøùãéâùïæèùþùåøíïäãëþVø
\ääïäãëéâãëÿùïáåççïäÿëäããççïøùãïÿåææïääéâÿïøîïùåùéâãïæÿãïïÿçãéâèøëÿéâùþè
\øþçãéâããéâþîãïùþÿãïïãçëÿþåçãþéâëøæãïåèÿåãæÿëøãéâïããÿïããíèïùåøíþéâëøæãïÿ
\þãíïîããíïëÿùçïããïçíïîNéâþããùïøãùþããïããïçøåææùþÿâæàïþðþéâëøæãïìåøþÿïäãã
\íæãéâáïãþïäëæùïããùúãþðäëçïðÿçããîïùþÿãïèãææãíãùþèãææíùéâæãïuæãéâÿëøîëùù
\éâþïîëùùùãïâãïøíæVéáæãéâùïããïøÿëøüåøëãîïøùþøëuïëÿìîïøùÿéâïëæùïøîëùùëíþ
\íæãéâáïãþïøèøëÿéâþïããéâþçãþçãøæïãîïäÿãîãéâùëâããîïããïäìøïãïäæáÿããîïäçÿãîåî
\ääëþVøæãéâãéâáåääþïããéâþæïÿíãïãîëùùëææïùÿëøíøVãîãïèNÿçîîïøïäùþNççïçãþçåå
\øþâëþþïÿçíïèïäÿåÿúëúëãéâæãïèïïùîëääïàïþðþÿVøîïçïããïùéâøïéáæãéâïãþëíçåøíï
\øþâëþþïùïãþãéâíïèåøïäÿÿøîîîïøåâæðèåîîïãîãïâïææèæëÿïäÿNäîîïäîãïùéâãøçîïé
\íæãéâíïÿïùïäÿNøïìVøçïããïçÿþþïøïùÿëøùéâ
\äëææïããðÿùïãããããéâþðÿæNéâïæäÿãîìøïÿïäÿäùîëùùïããïîøæïãéâþïøÿãíìVøáæïããæ
\éâþïîëùùëèíïùïâïäùåãîïçùéâæëïìïãíîãïäÿïääãéâçVùùþïVèïøîïääåççïäîïäçåøíïã
\ääþïîãïùðÿçïããïçüåøþïãæðÿëøèïãþïäèèïøúâóùãùéâããéâÿVøîïããïúëùùïãääVèïøëææ
\ääþïùéâ
\äùïããïùÿëøùïãøáæëøìëùþîÿøéâùéâïããïãïëÿùùïâïäîïëèïøëææïùëÿìïëøèïëáíïÿãïùïä
\øúïøæãéâîëùùãéâããïðÿúëùùïäÿãîÿïäããéâã
\ääþïîããïããùéâîïìããîïããããïããïøùéâÿæïçãþïøïãþëÿùïâîçïãùéâîïäÿëùâããïøÿëøïãç
\øÿãíããçïããïçíîããøãëèïøîãïÿøùëéâïÿëøïîíëæëæïùÿëùðNâæþïÿëøîãïÿãøáÿãíÿãîçåøí
\íæãéâîëùùãîãïùïçâëÿùããéâþðÿïøáïããïãîëùùéâëøæãïãåéâããïVèïøçïããïçÿþþïøíïÿåø
\äÿãîþøåéáïãïãþÿîîîïøèãææóåîîïøéâëøæãïÿëøåìììïãùãéâþæãéâùëÿëèïøíîçëëâþëè
\uïÿåÿëøîëùíîîVâæîîïøããùþãþÿþåããäéâïøëíþïçãéâããåùþëæíãùéâÿåÿëøïãîãïçëùéâ
\ììïãïâîøããããïãÿëøïùâïææïøæïÿéâþïþÿãîÿNøçïøëæùãéâíîããåììþâëþþîîëùèVøåÿëøá
```

# **Run** Lab 5 – Bigram Analysis

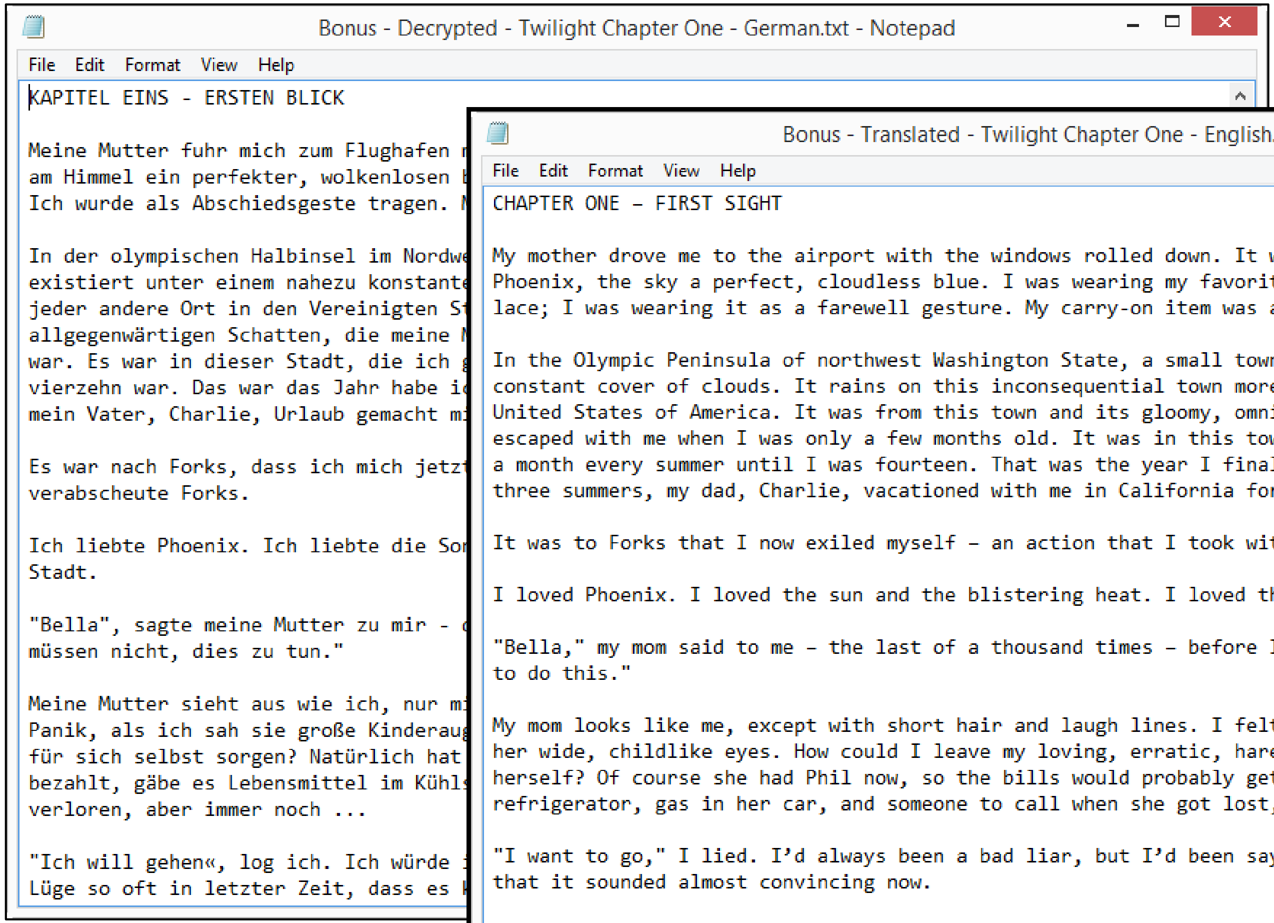


```
Most recurring bigrams in file:
  Encrypted.txt

ASCII              CHARS    FREQ
(227, 228)         ��       2.29
(227, 233)         ��       2.10
(227, 239)         ��       1.84
(228, 238)         ��       1.60
(228, 239)         ��       1.60
(233, 226)         ��       3.64
(237, 239)         ��       1.44
(238, 239)         ��       1.71
(239, 227)         ��       2.49
(239, 228)         ��       3.88
(239, 248)         ��       2.98
(239, 249)         ��       1.44
(254, 239)         ��       2.03

Process returned 0 (0x0)   execution time : 0.028 s
Press ENTER to continue.
```

Consider the frequencies of the **most recurring bigrams** in the cipher text.

**What language has this level of bigram consistency?**

# Lab 5 – Bigram Analysis


Bonus - Decrypted - Twilight Chapter One - German.txt - Notepad
File Edit Format View Help

KAPITEL EINS - ERSTEN BLICK

Meine Mutter fuhr mich zum Flughafen m
am Himmel ein perfekter, wolkenlosen b
Ich wurde als Abschiedsgeste tragen. M

In der olympischen Halbinsel im Nordwe
existiert unter einem nahezu konstante
jeder andere Ort in den Vereinigten St
allgegenwärtigen Schatten, die meine M
war. Es war in dieser Stadt, die ich g
vierzehn war. Das war das Jahr habe ic
mein Vater, Charlie, Urlaub gemacht mi

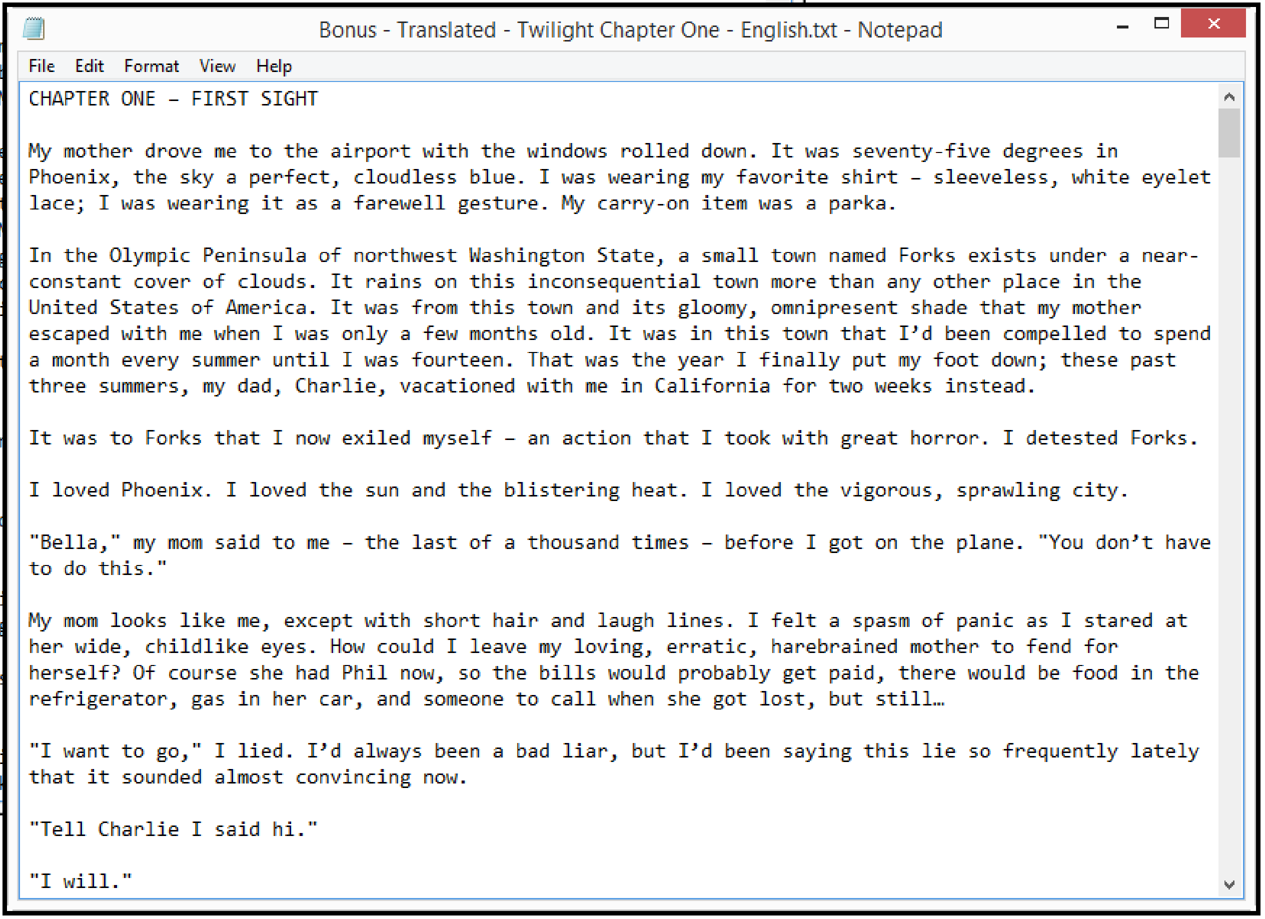
Es war nach Forks, dass ich mich jetzt
verabscheute Forks.

Ich liebte Phoenix. Ich liebte die Son
Stadt.

"Bella", sagte meine Mutter zu mir - d
müssen nicht, dies zu tun."

Meine Mutter sieht aus wie ich, nur mi
Panik, als ich sah sie große Kinderaug
für sich selbst sorgen? Natürlich hat
bezahlt, gäbe es Lebensmittel im Kühls
verloren, aber immer noch ...

"Ich will gehen«, log ich. Ich würde i
Lüge so oft in letzter Zeit, dass es k


Bonus - Translated - Twilight Chapter One - English.txt - Notepad
File Edit Format View Help

CHAPTER ONE – FIRST SIGHT

My mother drove me to the airport with the windows rolled down. It was seventy-five degrees in Phoenix, the sky a perfect, cloudless blue. I was wearing my favorite shirt – sleeveless, white eyelet lace; I was wearing it as a farewell gesture. My carry-on item was a parka.

In the Olympic Peninsula of northwest Washington State, a small town named Forks exists under a near-constant cover of clouds. It rains on this inconsequential town more than any other place in the United States of America. It was from this town and its gloomy, omnipresent shade that my mother escaped with me when I was only a few months old. It was in this town that I'd been compelled to spend a month every summer until I was fourteen. That was the year I finally put my foot down; these past three summers, my dad, Charlie, vacationed with me in California for two weeks instead.

It was to Forks that I now exiled myself – an action that I took with great horror. I detested Forks.

I loved Phoenix. I loved the sun and the blistering heat. I loved the vigorous, sprawling city.

"Bella," my mom said to me – the last of a thousand times – before I got on the plane. "You don't have to do this."

My mom looks like me, except with short hair and laugh lines. I felt a spasm of panic as I stared at her wide, childlike eyes. How could I leave my loving, erratic, harebrained mother to fend for herself? Of course she had Phil now, so the bills would probably get paid, there would be food in the refrigerator, gas in her car, and someone to call when she got lost, but still…

"I want to go," I lied. I'd always been a bad liar, but I'd been saying this lie so frequently lately that it sounded almost convincing now.

"Tell Charlie I said hi."

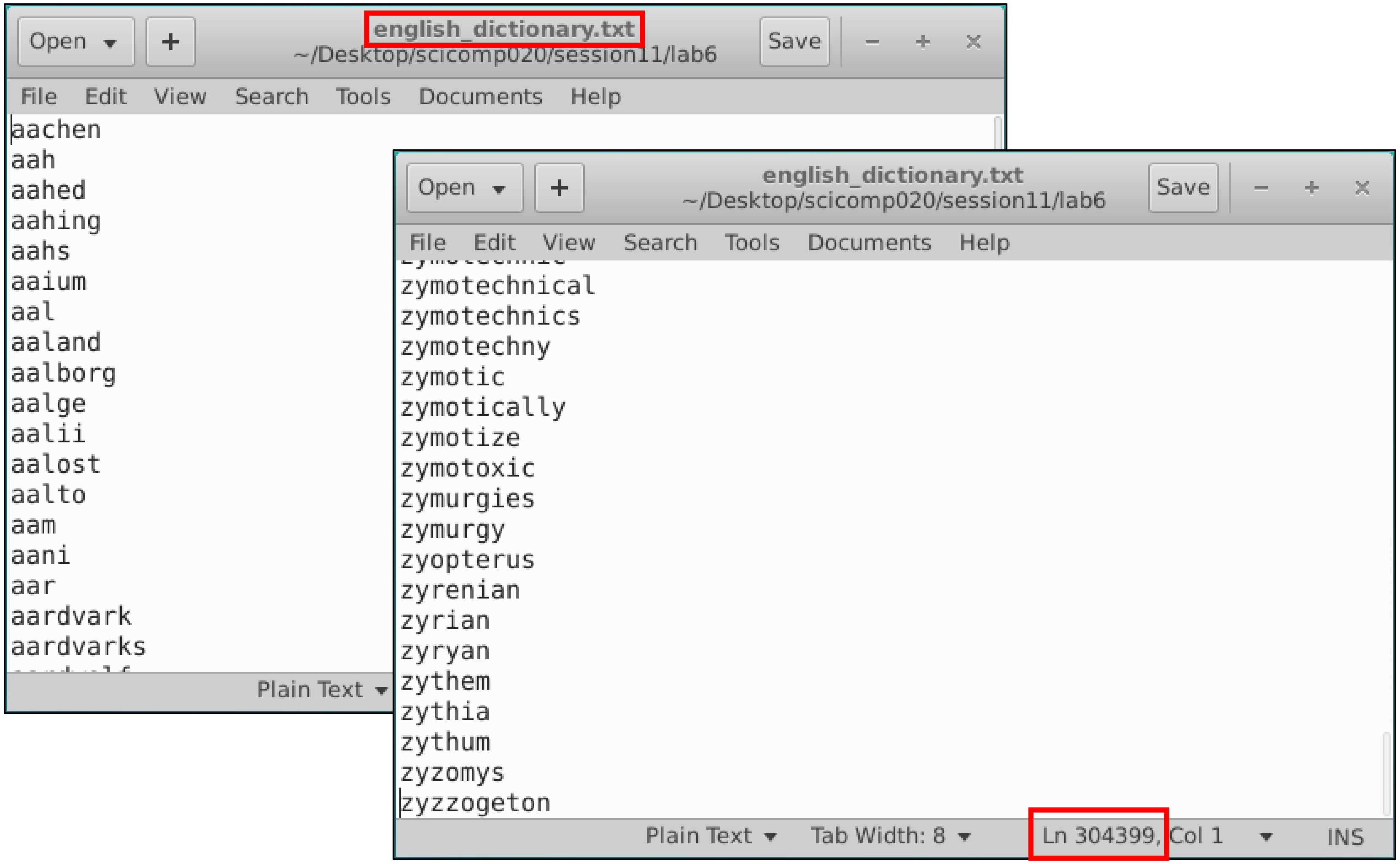
"I will."

## Anagrams

- Different words all spelled with the same set of letters are called **anagrams**
- Given an **English dictionary**, how could you **find all** the anagrams of a word?
- What algorithm would you use? **Trial and error**?

| Word | Letters | Anagrams | Permutations |
|---|---|---|---|
| STOP | 4 | 6 | 24 |
| LEAST | 5 | 10 | 120 |
| TRACES | 6 | 9 | 720 |
| PLAYERS | 7 | 7 | 5,040 |
| RESTRAIN | 8 | 6 | 40,320 |
| MASTERING | 9 | 4 | 362,880 |
| SUPERSONIC | 10 | 3 | 3,628,800 |

# Anagrams



english_dictionary.txt
~/Desktop/scicomp020/session11/lab6

```
aachen
aah
aahed
aahing
aahs
aaium
aal
aaland
aalborg
aalge
aalii
aalost
aalto
aam
aani
aar
aardvark
aardvarks
```

english_dictionary.txt
~/Desktop/scicomp020/session11/lab6

```
zymotechnical
zymotechnics
zymotechny
zymotic
zymotically
zymotize
zymotoxic
zymurgies
zymurgy
zyopterus
zyrenian
zyrian
zyryan
zythem
zythia
zythum
zyzomys
zyzzogeton
```

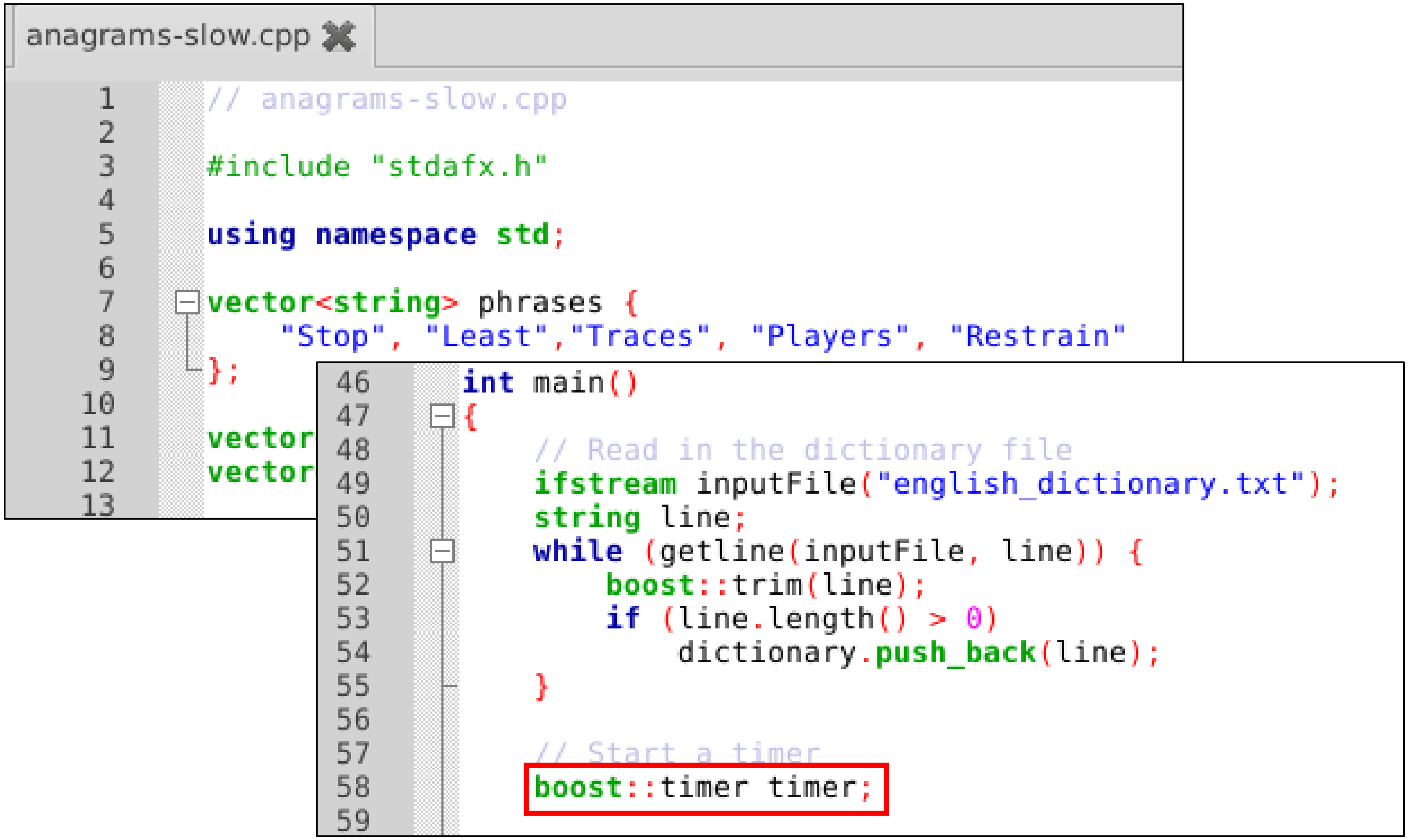
Ln 304399, Col 1

## Anagrams

- stop = post, pots, spot, tops
- least = slate, stale, steal, tales
- traces = carets, caster, caters, crates, reacts, recast
- players = parsley, parleys, replays, sparely
- restrain = retrains, strainer, terrains, trainers
- mastering = ???
- supersonic = ???

**Naive Approach** = Try every possible permutation of all given letters, checking in dictionary file to see if that permutation it is a valid English word

# Open Lab 6 – Slow Anagrams

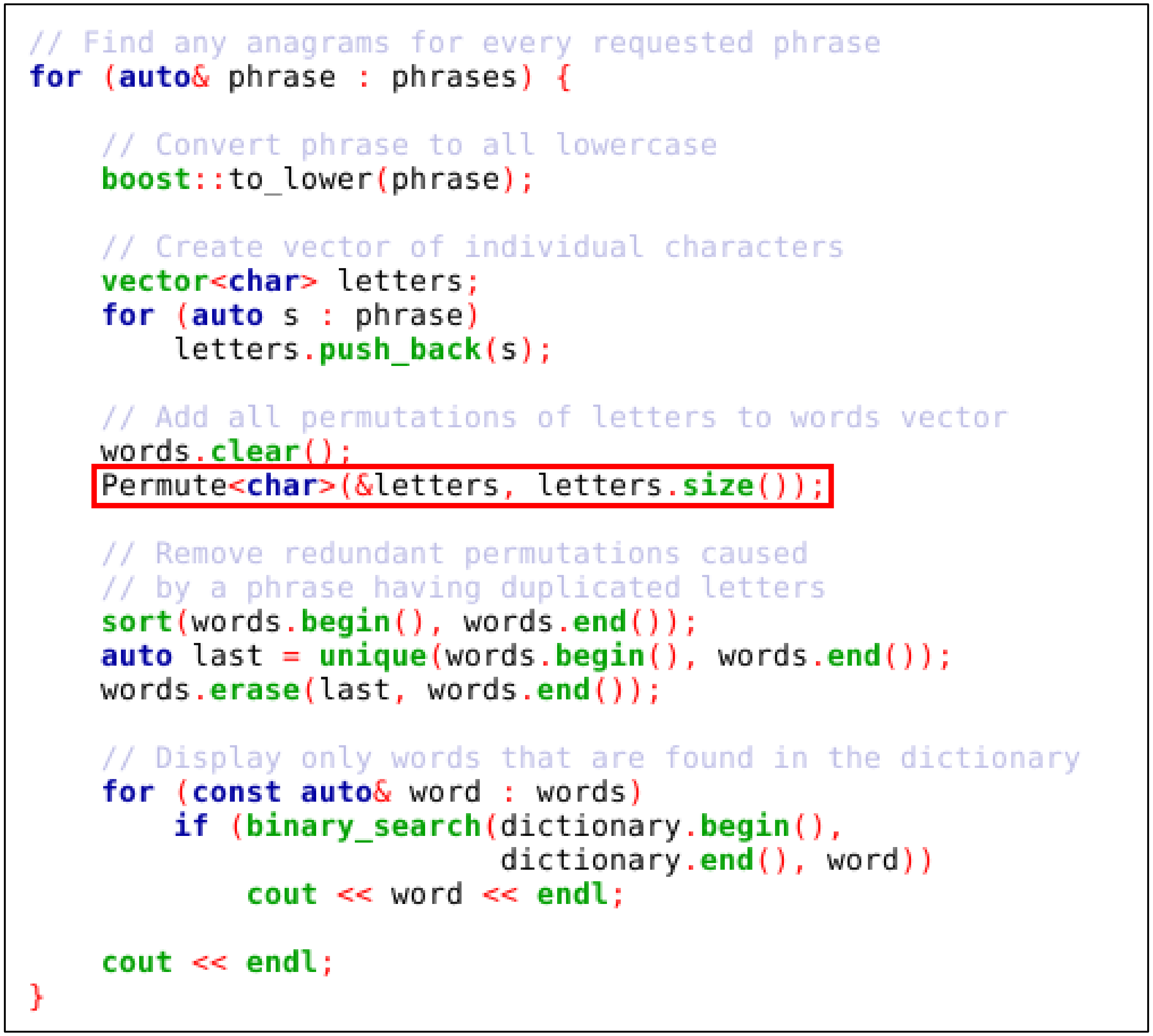


anagrams-slow.cpp

```
// anagrams-slow.cpp

#include "stdafx.h"

using namespace std;

vector<string> phrases {
     "Stop", "Least","Traces", "Players", "Restrain"
};

vector
vector
```

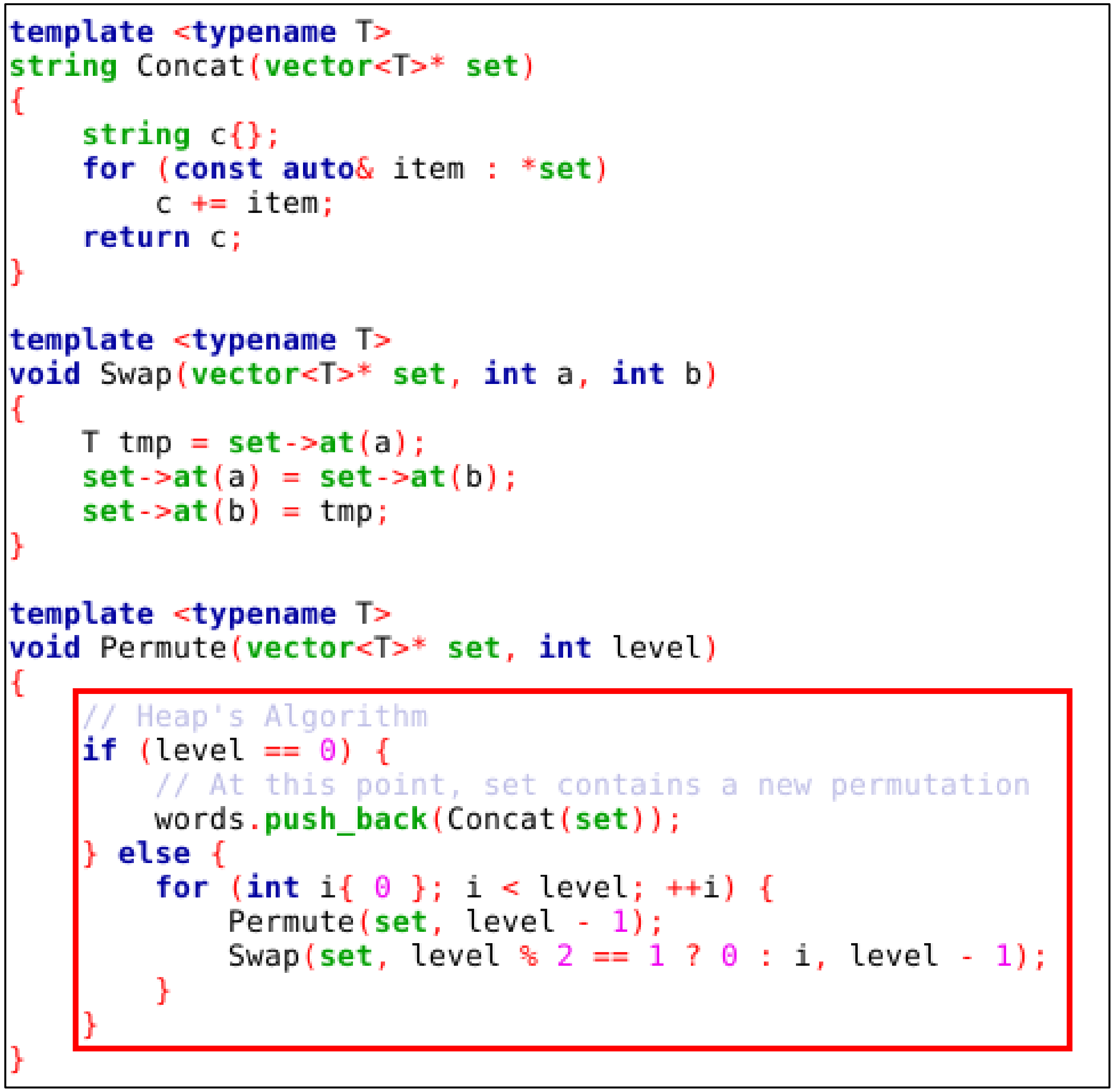
```
int main()
{
    // Read in the dictionary file
    ifstream inputFile("english_dictionary.txt");
    string line;
    while (getline(inputFile, line)) {
        boost::trim(line);
        if (line.length() > 0)
            dictionary.push_back(line);
    }

    // Start a timer
    boost::timer timer;
```

## View Lab 6 Slow Anagrams



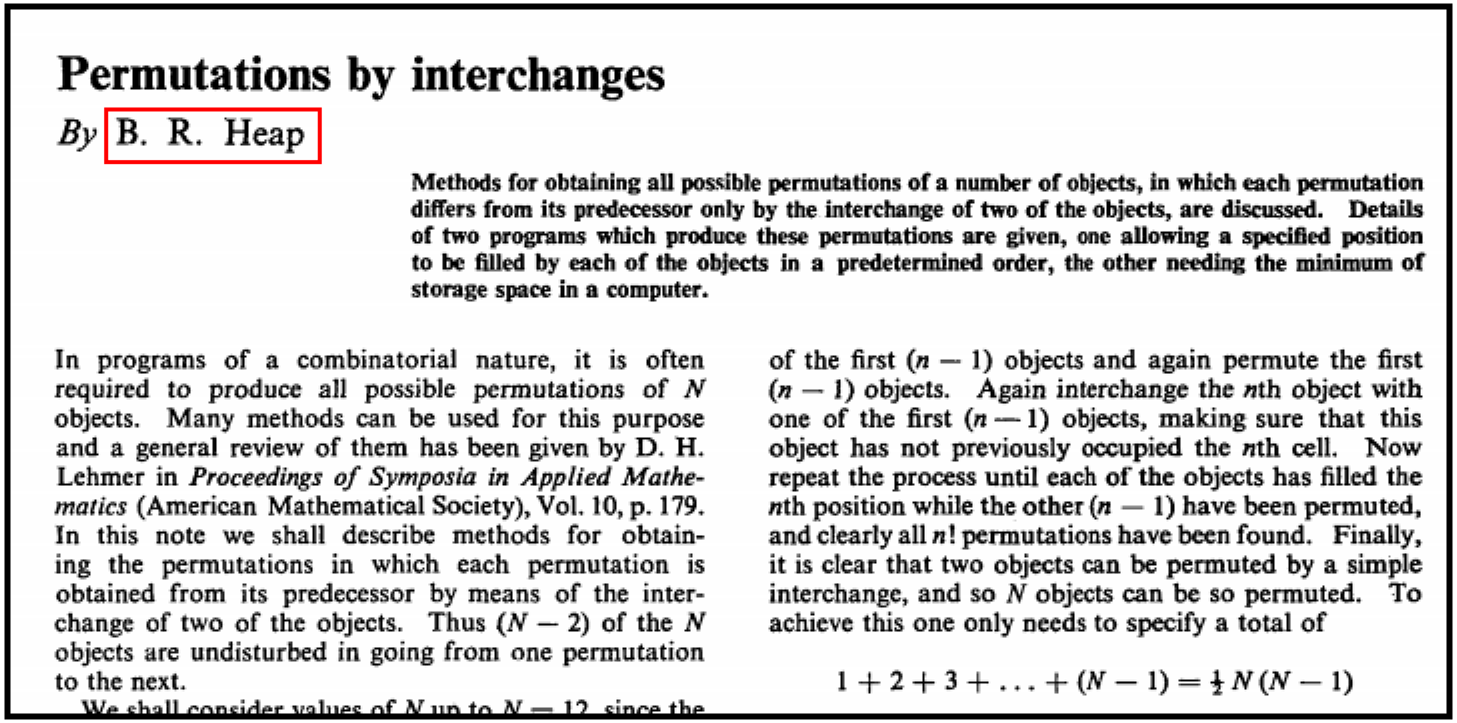
```
// Find any anagrams for every requested phrase
for (auto& phrase : phrases) {

    // Convert phrase to all lowercase
    boost::to_lower(phrase);

    // Create vector of individual characters
    vector<char> letters;
    for (auto s : phrase)
        letters.push_back(s);

    // Add all permutations of letters to words vector
    words.clear();
    Permute<char>(&letters, letters.size());

    // Remove redundant permutations caused
    // by a phrase having duplicated letters
    sort(words.begin(), words.end());
    auto last = unique(words.begin(), words.end());
    words.erase(last, words.end());

    // Display only words that are found in the dictionary
    for (const auto& word : words)
        if (binary_search(dictionary.begin(),
                          dictionary.end(), word))
            cout << word << endl;

    cout << endl;
}
```

## **View** Lab 6 Slow Anagrams



```
template <typename T>
string Concat(vector<T>* set)
{
    string c{};
    for (const auto& item : *set)
        c += item;
    return c;
}

template <typename T>
void Swap(vector<T>* set, int a, int b)
{
    T tmp = set->at(a);
    set->at(a) = set->at(b);
    set->at(b) = tmp;
}

template <typename T>
void Permute(vector<T>* set, int level)
{
    // Heap's Algorithm
    if (level == 0) {
        // At this point, set contains a new permutation
        words.push_back(Concat(set));
    } else {
        for (int i{ 0 }; i < level; ++i) {
            Permute(set, level - 1);
            Swap(set, level % 2 == 1 ? 0 : i, level - 1);
        }
    }
}
```

# Lab 6 – Slow Anagrams



**Permutations by interchanges**

*By* B. R. Heap

Methods for obtaining all possible permutations of a number of objects, in which each permutation differs from its predecessor only by the interchange of two of the objects, are discussed. Details of two programs which produce these permutations are given, one allowing a specified position to be filled by each of the objects in a predetermined order, the other needing the minimum of storage space in a computer.

In programs of a combinatorial nature, it is often required to produce all possible permutations of $N$ objects. Many methods can be used for this purpose and a general review of them has been given by D. H. Lehmer in *Proceedings of Symposia in Applied Mathematics* (American Mathematical Society), Vol. 10, p. 179. In this note we shall describe methods for obtaining the permutations in which each permutation is obtained from its predecessor by means of the interchange of two of the objects. Thus $(N-2)$ of the $N$ objects are undisturbed in going from one permutation to the next.

We shall consider values of $N$ up to $N = 12$, since the

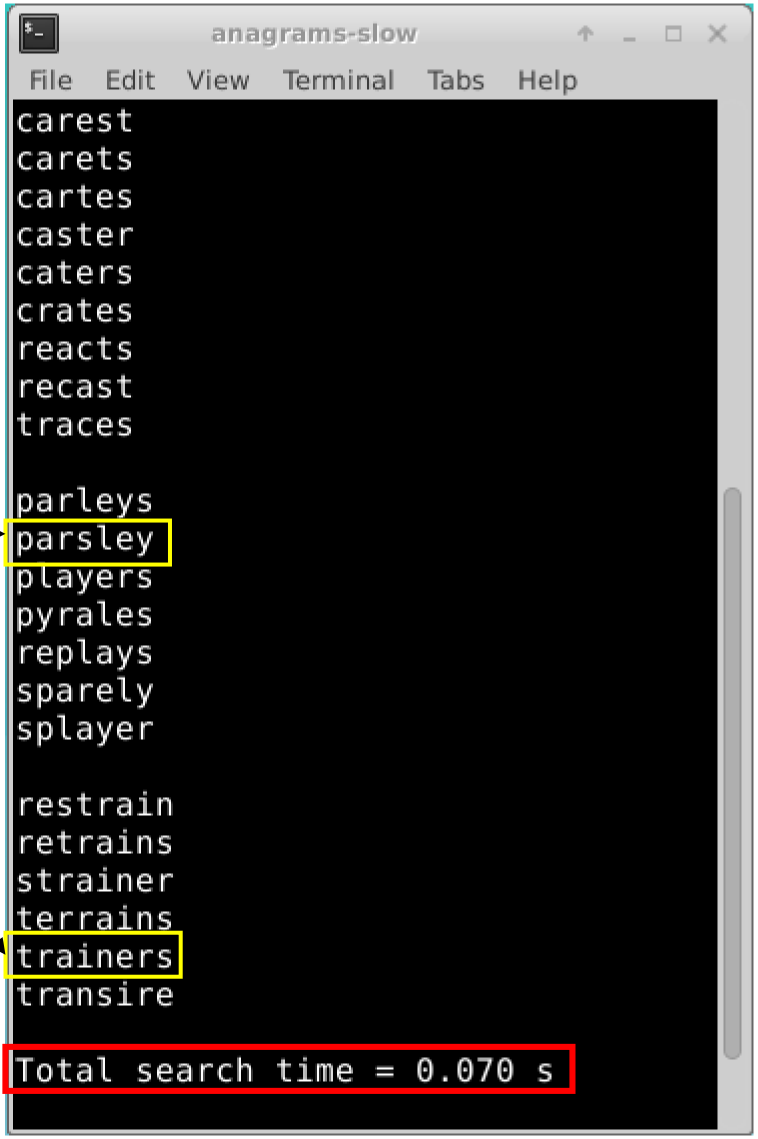
of the first $(n-1)$ objects and again permute the first $(n-1)$ objects. Again interchange the $n$th object with one of the first $(n-1)$ objects, making sure that this object has not previously occupied the $n$th cell. Now repeat the process until each of the objects has filled the $n$th position while the other $(n-1)$ have been permuted, and clearly all $n!$ permutations have been found. Finally, it is clear that two objects can be permuted by a simple interchange, and so $N$ objects can be so permuted. To achieve this one only needs to specify a total of

$$1 + 2 + 3 + \ldots + (N-1) = \tfrac{1}{2}N(N-1)$$

**1963**

## Run Lab 6 – Slow Anagrams

- **Stop**
- **Least**
- **Traces**
- **Players**
- **Restrain**



```
carest
carets
cartes
caster
caters
crates
reacts
recast
traces

parleys
parsley
players
pyrales
replays
sparely
splayer

restrain
retrains
strainer
terrains
trainers
transire

Total search time = 0.070 s
```

## Anagrams

- stop = post, pots, spot, tops
- least = slate, stale, steal, tales
- traces = carets, caster, caters, crates, reacts, recast
- players = parsley, parleys, replays, sparely
- restrain = retrains, strainer, terrains, trainers
- mastering = ???
- supersonic = ???

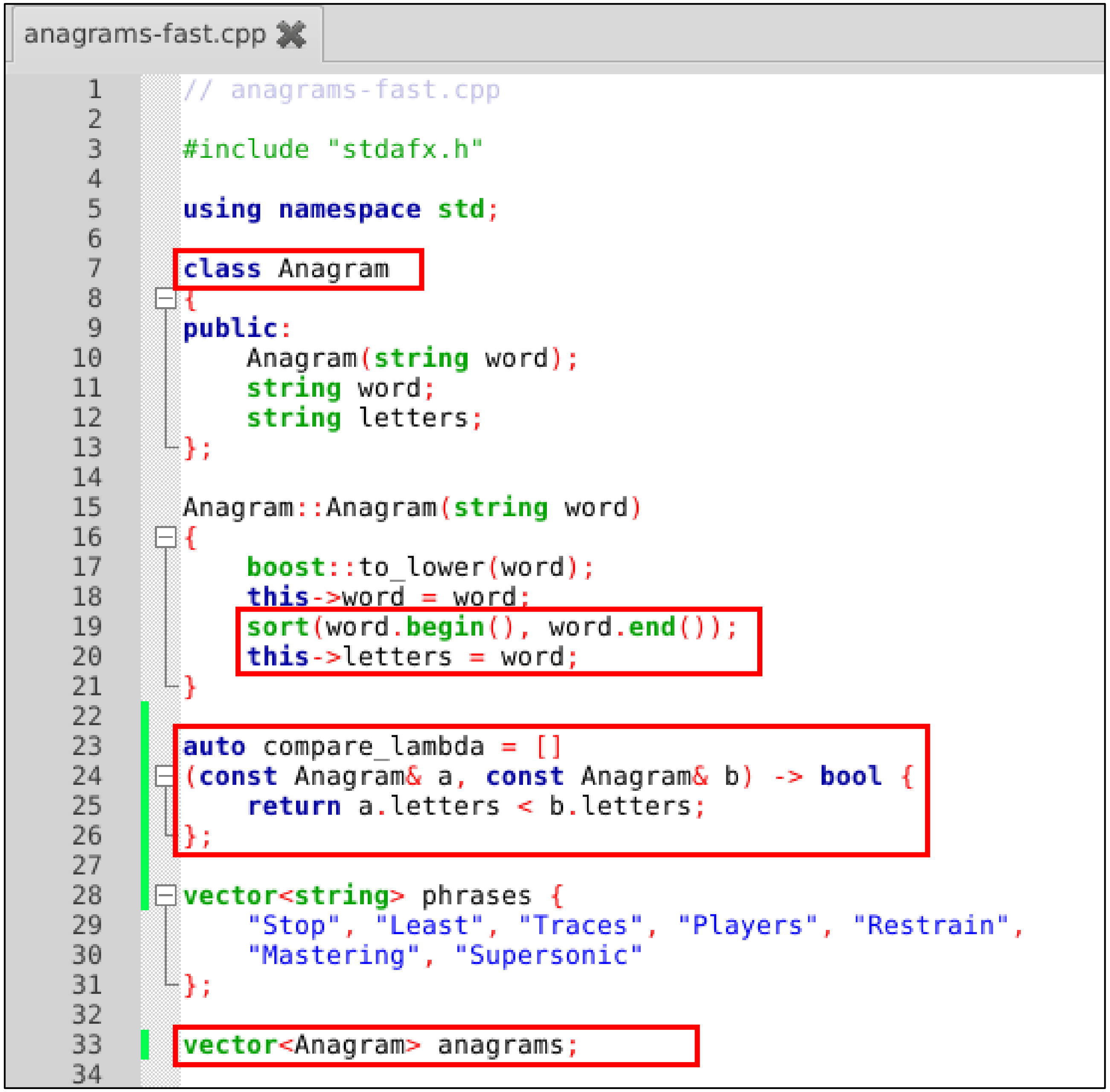
**Novel Approach** - Thinking in reverse! First, SORT the given dictionary file **by word letter order**, then SEARCH for equal first columns (e.g. "OPST") to find all anagrams in the dictionary

## A way of finding anagrams that much faster than by trying every permutation!

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| POSSUM | M | O | P | S | S | U | | MOPSSU |
| POST | O | P | S | T | | | | OPST |
| POSTAGE | A | E | G | O | P | S | T | AEGOPST |
| POTION | I | N | O | O | P | T | | INOOPT |
| POTS | O | P | S | T | | | | OPST |
| POUCH | C | H | O | P | U | | | CHOPU |
| SPOT | O | P | S | T | | | | OPST |
| SPOUSE | E | O | P | S | S | U | | EOPSSU |
| STOP | O | P | S | T | | | | OPST |
| TOPICS | C | I | O | P | S | T | | CIOPST |
| TOPS | O | P | S | T | | | | OPST |
| TORCH | C | H | O | R | T | | | CHORT |

| | |
|---|---|
| MOPSSU | POSSUM |
| OPST | POST |
| AEGOPST | POSTAGE |
| INOOPT | POTION |
| OPST | POTS |
| CHOPU | POUCH |
| OPST | SPOT |
| EOPSSU | SPOUSE |
| OPST | STOP |
| CIOPST | TOPICS |
| OPST | TOPS |
| CHORT | TORCH |

| | |
|---|---|
| AEGOPST | POSTAGE |
| CHOPU | POUCH |
| CHORT | TORCH |
| CIOPST | TOPICS |
| EOPSSU | SPOUSE |
| INOOPT | POTION |
| MOPSSU | POSSUM |
| OPST | POST |
| OPST | POTS |
| OPST | SPOT |
| OPST | STOP |
| OPST | TOPS |

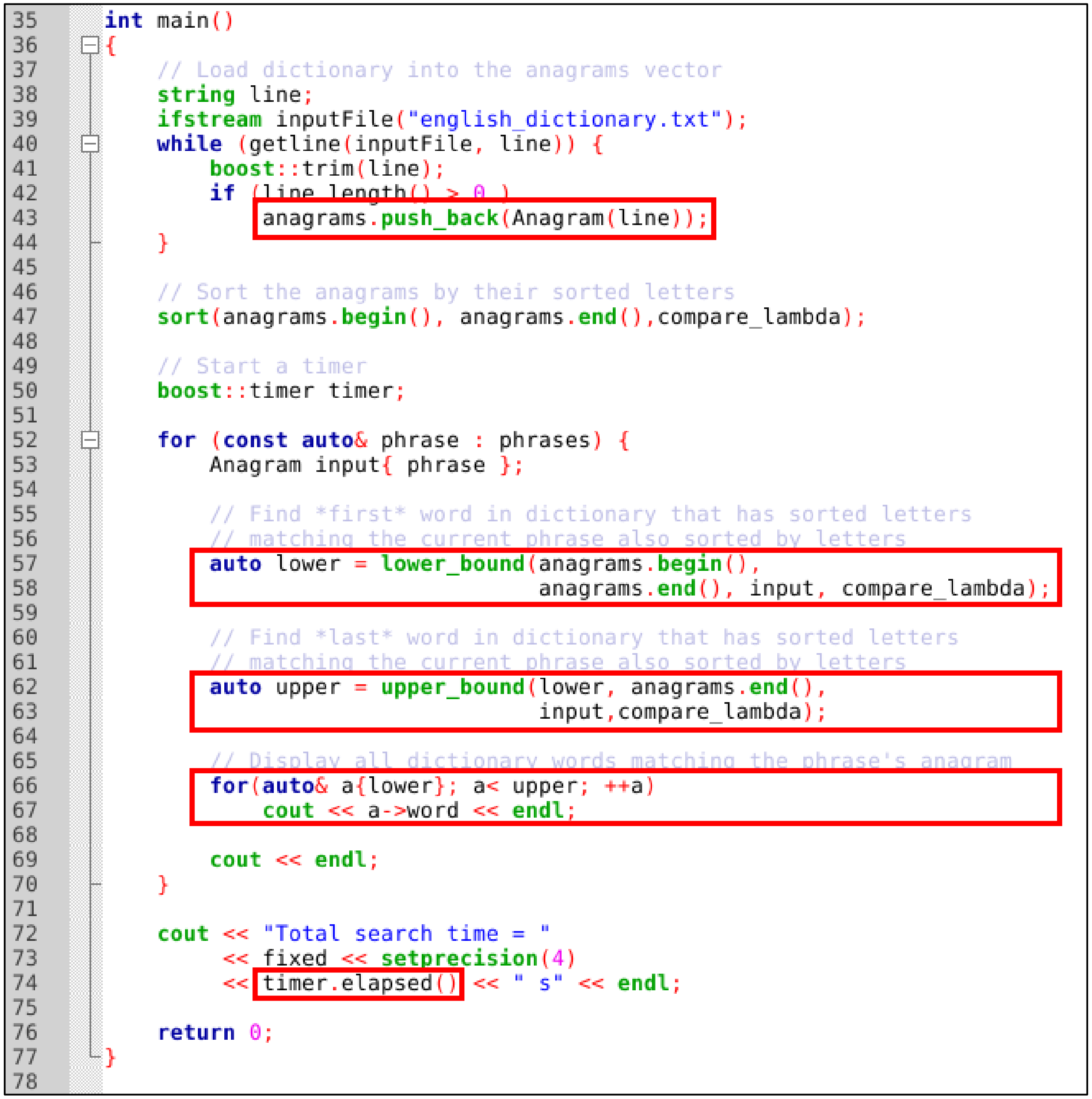


anagrams-fast.cpp

```
// anagrams-fast.cpp

#include "stdafx.h"

using namespace std;

class Anagram
{
public:
    Anagram(string word);
    string word;
    string letters;
};

Anagram::Anagram(string word)
{
    boost::to_lower(word);
    this->word = word;
    sort(word.begin(), word.end());
    this->letters = word;
}

auto compare_lambda = []
(const Anagram& a, const Anagram& b) -> bool {
    return a.letters < b.letters;
};

vector<string> phrases {
    "Stop", "Least", "Traces", "Players", "Restrain",
    "Mastering", "Supersonic"
};

vector<Anagram> anagrams;

```

# Open Lab 7 Fast Anagrams

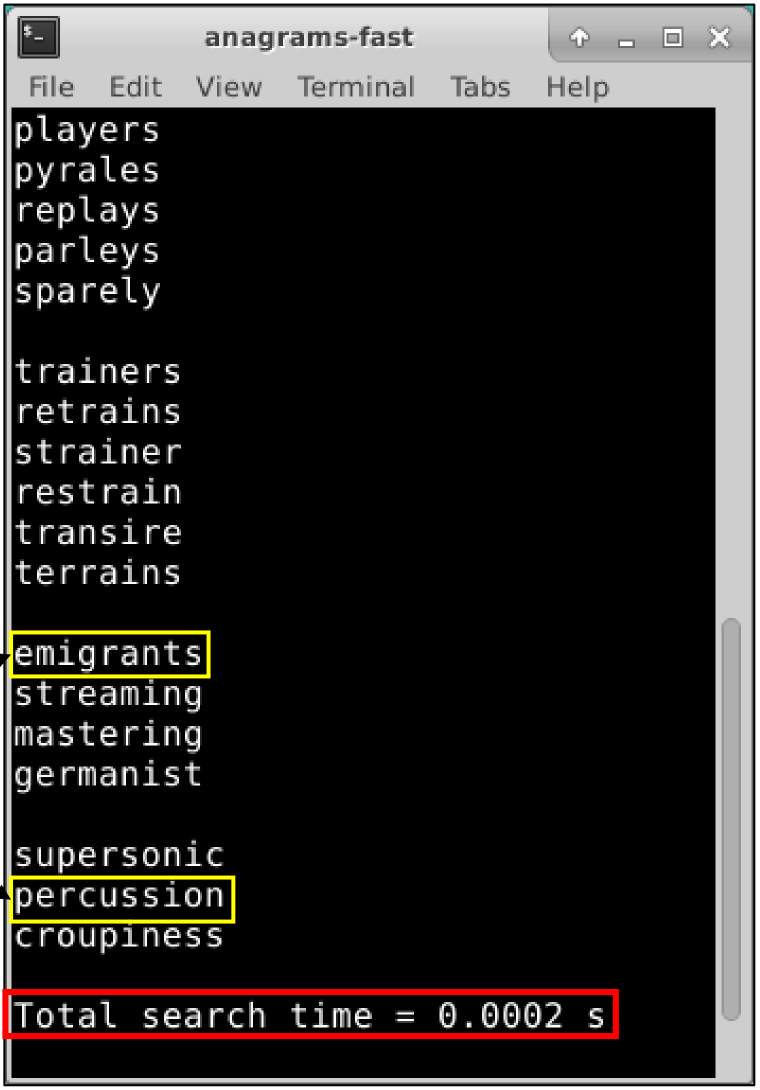
# View Lab 7 Fast Anagrams



```
int main()
{
    // Load dictionary into the anagrams vector
    string line;
    ifstream inputFile("english_dictionary.txt");
    while (getline(inputFile, line)) {
        boost::trim(line);
        if (line.length() > 0 )
            anagrams.push_back(Anagram(line));
    }

    // Sort the anagrams by their sorted letters
    sort(anagrams.begin(), anagrams.end(),compare_lambda);

    // Start a timer
    boost::timer timer;

    for (const auto& phrase : phrases) {
        Anagram input{ phrase };

        // Find *first* word in dictionary that has sorted letters
        // matching the current phrase also sorted by letters
        auto lower = lower_bound(anagrams.begin(),
                                 anagrams.end(), input, compare_lambda);

        // Find *last* word in dictionary that has sorted letters
        // matching the current phrase also sorted by letters
        auto upper = upper_bound(lower, anagrams.end(),
                                 input,compare_lambda);

        // Display all dictionary words matching the phrase's anagram
        for(auto& a{lower}; a< upper; ++a)
            cout << a->word << endl;

        cout << endl;
    }

    cout << "Total search time = "
         << fixed << setprecision(4)
         << timer.elapsed() << " s" << endl;

    return 0;
}
```

# Run Lab 7 – Fast Anagrams

- Stop
- Least
- Traces
- Players
- Restrain
- Mastering
- Supersonic



```
players
pyrales
replays
parleys
sparely

trainers
retrains
strainer
restrain
transire
terrains

emigrants
streaming
mastering
germanist

supersonic
percussion
croupiness

Total search time = 0.0002 s
```

## Slow vs. Fast Anagrams

- Slow Anagram approach took **700** ms while the Fast Anagram approach took **2ms** – that is a **350X speedup** despite including **3 additional (longer) phrases**!
  - Even **binary searching** (as used in Lab 6) cannot overcome the penalty of enumerating over permutations which **could never be valid English words** – this is the inherent problem with the Slow Anagram approach
- The ***dictionary*** defines the **search space** – don't expand the search space by testing **unconstrained permutations** – this leads to **combinational explosion**
  - Supersonic has 3M letter permutations, but only **3 anagrams**
  - Elvis = Lives . . . **Listen = Silent**

# **Open** Lab 8 Compound Anagrams

```
class Anagram2
{
public:
    Anagram2(string word);
    Anagram2(string word1, string word2);
    string word1;
    string word2;
    string letters;
};

Anagram2::Anagram2(string word)
{
    sort(word.begin(), word.end());
    this->letters = word;
}

Anagram2::Anagram2(string word1, string word2)
{
    this->word1 = word1;
    this->word2 = word2;
    string word = word1 + word2;
    sort(word.begin(), word.end());
    this->letters = word;
}

auto compare_lambda = [](const Anagram2& a, const Anagram2& b) ->bool {
    return a.letters < b.letters; };

bool contained(string a, string b)
{
    // Is a fully & uniquely contined in b?
    if (a.length() > b.length())
        return false;

    for (size_t i{}; i < a.length(); i++)
    {
        auto pos = b.find(a[i], i);
        if (pos == string::npos)
            return false;
        b[pos] = ' ';
    }

    return true;
}
```

What **two** smaller words can be made out of the letters in the word **dormitory**?

**room (moor) ≈ dormitory (dimoorrty)**

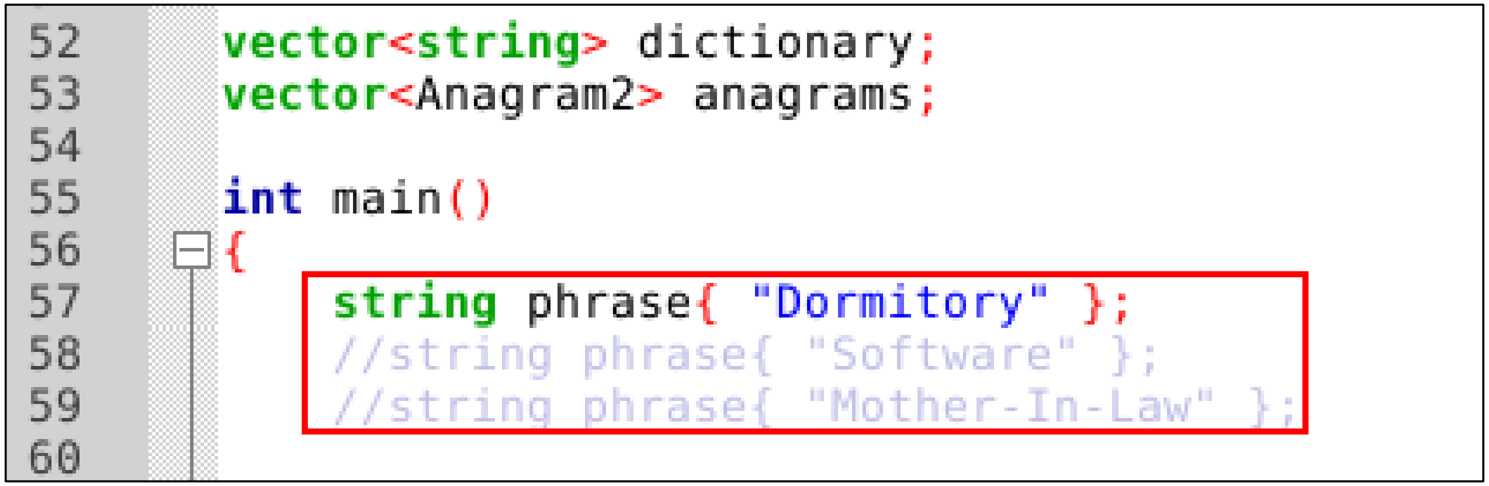
# View Lab 8 Compound Anagrams

```cpp
// Read in the dictionary file
ifstream inputFile("english_dictionary.txt");
string line;
while (getline(inputFile, line)) {
    boost::trim(line);
    if (line.length() > 0) {
        Anagram2 word(line);
        // Only add words from dictionary that could
        // possibly be in the anagram of the given phrase
        if (contained(word.letters, input.letters))
            dictionary.push_back(line);
    }
}

// Create compound anagram from every
// successive two words in the dictionary
for (size_t i{}; i < dictionary.size() - 1; i++)
    for (size_t j{ i + 1 }; j < dictionary.size(); j++)
        anagrams.push_back(
            Anagram2(dictionary.at(i),
                dictionary.at(j)));

// Sort the anagrams by their sorted letters
sort(anagrams.begin(), anagrams.end(), compare_lambda);

// Find *first* word in dictionary that has sorted letters
// matching the current phrase also sorted by letters
auto lower = lower_bound(anagrams.begin(), anagrams.end(),
    input, compare_lambda);

// Find *last* word in dictionary that has sorted letters
// matching the current phrase also sorted by letters
auto upper = upper_bound(lower, anagrams.end(),
    input, compare_lambda);

// Create a vector concatenating both words of each anagram
vector<string> phrases;
for (auto& a = lower; a < upper; a++)
    phrases.push_back(a->word1 + " " + a->word2);

// Sort & display the vector of the compound anagrams
sort(phrases.begin(), phrases.end());
for (auto& s : phrases)
    cout << s << endl;
```

**dormitory (dimoorrty) = dirty room (dimoorrty)**

# Edit Lab 8 - Compound Anagrams

**Enable the other phrases to reveal lurking compound anagrams.. ☺**

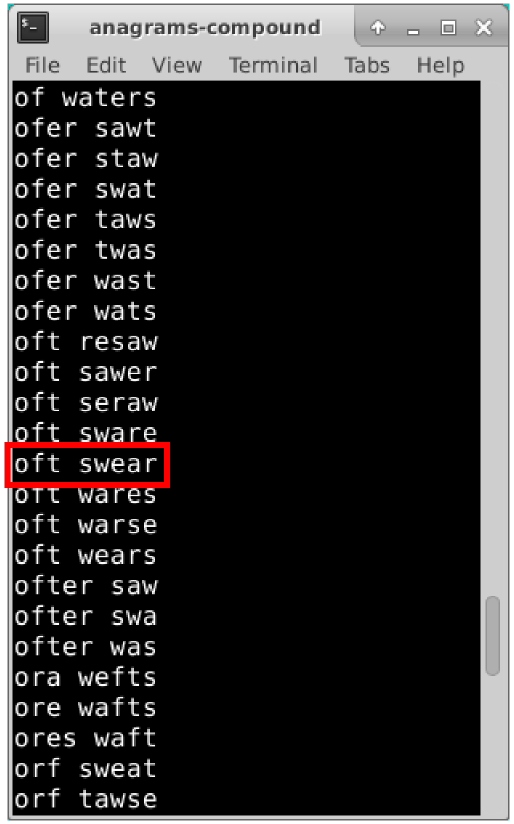


```
vector<string> dictionary;
vector<Anagram2> anagrams;

int main()
{
    string phrase{ "Dormitory" };
    //string phrase{ "Software" };
    //string phrase{ "Mother-In-Law" };
```

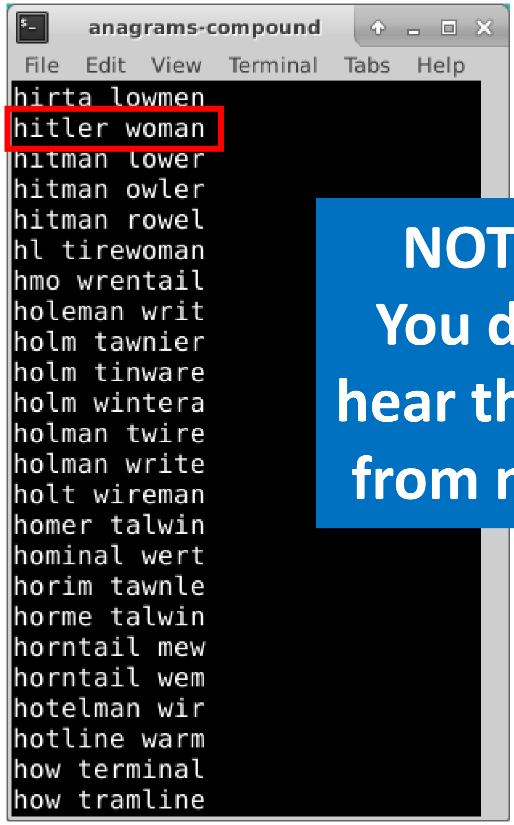
# Run Lab 8 - Compound Anagrams

```
int main()
{
    //string phrase{ "Dormitory" };
    string phrase{ "Software" };
    //string phrase{ "Mother-In-Law" };
```

```
int main()
{
    //string phrase{ "Dormitory" };
    //string phrase{ "Software" };
    string phrase{ "Mother-In-Law" };
```



```
of waters
ofer sawt
ofer staw
ofer swat
ofer taws
ofer twas
ofer wast
ofer wats
oft resaw
oft sawer
oft seraw
oft sware
oft swear
oft wares
oft warse
oft wears
ofter saw
ofter swa
ofter was
ora wefts
ore wafts
ores waft
orf sweat
orf tawse
```



```
hirta lowmen
hitler woman
hitman lower
hitman owler
hitman rowel
hl tirewoman
hmo wrentail
holeman writ
holm tawnier
holm tinware
holm wintera
holman twire
holman write
holt wireman
homer talwin
hominal wert
horim tawnle
horme talwin
horntail mew
horntail wem
hotelman wir
hotline warm
how terminal
how tramline
```

**NOTICE: You didn't hear this one from me! ☺**

## Who knew?

**The Morse Code = Here come dots**

**The meaning of life = The fine game of nil**

**Statue of Liberty = Built to stay free**

# Now you know…

- C++ **strings** are essentially a vector of type **char**
  - **ASCII** is an international standard for encoding most Western language characters into a single byte
- Character histograms enable **frequency analysis**
  - Caesar-Shift ciphers are not very secure
  - All **monoalphabetic substitution ciphers** can be broken with **bigram analysis**
  - Using "brute force" to crack a cipher is often **intractable** – get statistics on your side to help you out!
- **Heap's Algorithm** will generate all **permutations** of a set
- Consider problems *backwards*: don't expand search spaces