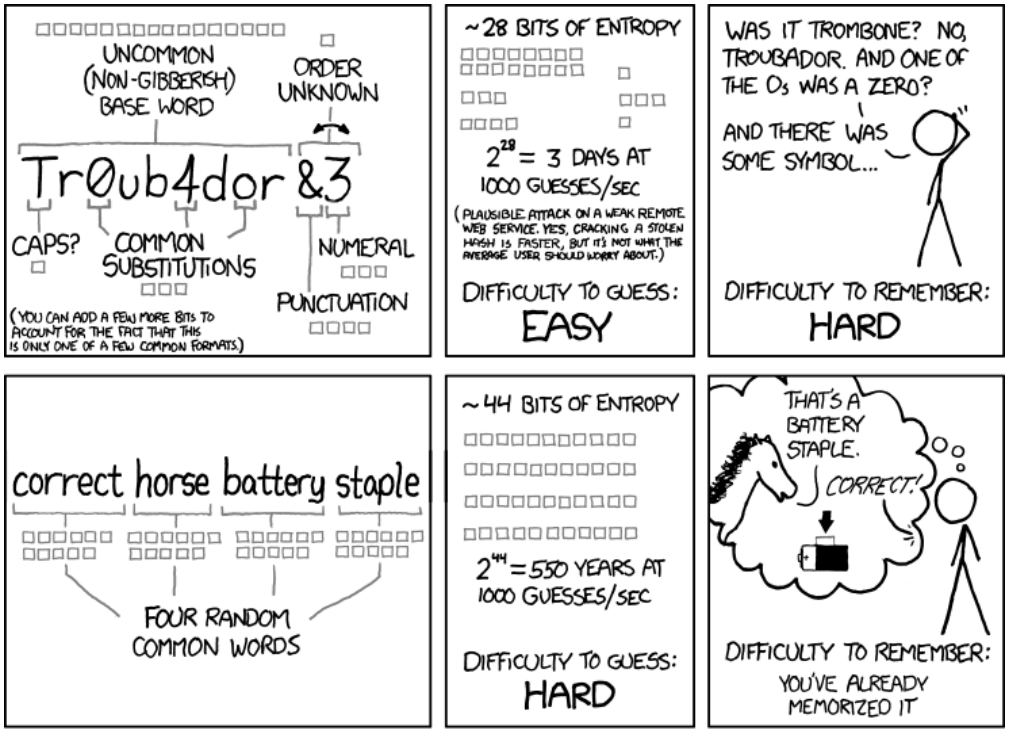
CS319:
Scientific Computing
(with C++)

Writing a password frequency analyser (as an excuse to study `templates` and some sorting algorithms)

Week 6:
9am and 4pm, 15 Feb 2017



http://xkcd.com/936

About Lab 4

- This is an open-ended task: at the very least your code should
  - Check if there are the same number of left braces as right braces;
  - Check if every right brace corresponds to a left brace;

  But there are any improvements possible...
- Carefully document the capabilities and limitations of your code. For example, it is desirable that your program correctly parse comments. But if it doesn't, that should be included in the documentation.
- **At the very least**, include your name and ID number in any source file.
- Where necessary, include comments that describe what the program is doing. However, try to make the program "self documenting" as possible by using descriptive function and variable names.
- Make your program as robust as possible.
- Before submitting, check the rubric to you are sure you understand the grading scheme. Also, **let me know if you think the rubrics could be improved**.
- When you open a file, check that it opened correctly.
- Test your code against the six test cases at `http://www.maths.nuigalway.ie/~niall/CS319/lab4`

1. Our aim: analyse a large list of passwords
2. Our algorithm (high-level)
3. Templates
   - Function Templates
4. Merge Sort
   - Why is Merge Sort faster than Bubble?
5. Analysing the passwords file
6. Sorting: A note on complexity

Our aim: analyse a large list of passwords

Our aim today is to take a very long list of passwords and to work out which is the most common.

The source of the data is the infamous **RockYou** password file, a list of over 30,000,000 unencrypted passwords stolen from RockYou in 2009 ← link!, and now widely available online. The version we'll work with was provided by a David Malone[1] from NUI Maynooth, who used it when writing an article Investigating the Distribution of Password Choices[2] ← link!

The uses of such files include optimising dictionary attacks (see notes from class that explain why).

The benign reasons for wanting to do this include

- We could use this as a way of testing the security of our own systems;
- Understanding how these attacks are done help up protect against them.

[1] Unrelated to today's class, David also wrote a cool article *To what does the harmonic series converge?* (Irish Mathematical Society Bulletin. Summer 2013, 71, 59–66).

[2] David Malone and Kevin Maher. *Investigating the Distribution of Password Choices*. International conference on the World Wide Web (WWW). 19 April 2012.

Our algorithm (high-level)

Given a list of 30,000,000 passwords, how shall we work out which 10 (say) occur most frequently?

Templates

We have now worked out that we need to do some list sorting. Presently, we'll recap on a sorting function that we used in Week 3. However, it just sorted integers. We'll need to sort list of strings, or perhaps lists of objects belonging to a class we define. So we would like to write a `sort` function that works for **any** datatype.

If we took our old `Sort(int *list, int length)` function (from `Week03/04Sort.cpp`), we could rewrite it for (say) `string`s: `Sort(string *list, int length)`

Most of the source code of the two functions would be identical: we'd just replace several instances of the datatype `int` with `string`.

To avoid this repetition, and to allow us to write functions or class **generic** datatypes, C++ provides **template**s.

Today we will only consider **function templates**. We'll return to the related idea of **class templates** another time.

To perform essentially identical operations for different types of data compactly, use function templates.

- Syntax: `template <typename T>` immediately precedes the function definition. It means that we'll be referring to the generic datatype as `T` in the function definition.
- Write a single function template definition. In it, the generic datatype is named `T`.
- Based on the argument types provided in calls to the function, the compiler automatically creates functions to handle each type of call appropriately.

In the example below, which you can find in detail in `01FunctionTemplate.cpp`, we'll write three functions:

(a) `PrintList(MyType *x, int n)`

(b) `void Sort(MyType &a, MyType &b)`

(c) `void Sort(MyType *x, int n)`

The function declarations:

01FunctionTemplate.cpp

```
template <typename MyType>
void PrintList(MyType *x, unsigned int n);

template <typename MyType>
void Sort(MyType &a, MyType &b);

template <typename MyType>
void Sort(MyType *list, unsigned int length);
```

The (bubble) `Sort` functions:

01FunctionTemplate.cpp

```
template <typename MyType>
void Sort(MyType &a, MyType &b) {
  if (a>b)
  {
    MyType tmp;
    tmp=a;
    a=b;
    b=tmp;
  }
}
```

```
template <typename MyType>
void Sort(MyType *x, unsigned int n) {
  int i, k;
  for (i=n-1; i>1; i--)
    for (k=0; k<i; k++)
      Sort(x[k], x[k+1]);
}
```

01FunctionTemplate.cpp

```
22  int main(void )
    {
24    int Numbers[8];
      char Letters[8];
26
      for (int i=0; i<8; i++)
28      Numbers[i]=rand()%40;

30    for (int i=0; i<8; i++)
        Letters[i]='A'+rand()%26;
```

01FunctionTemplate.cpp

```
    std::cout << "Before sorting:" << std::std::endl;
34  std::cout << "Numbers: "; PrintList(Numbers, 8);
    std::cout << "Letters: "; PrintList(Letters, 8);
36
    Sort(Numbers, 8);
38  Sort(Letters, 8);

40  std::cout << "After sorting: " << std::std::endl;
    std::cout << "Numbers: "; PrintList(Numbers, 8);
42  std::cout << "Letters: "; PrintList(Letters, 8);
```

Typical output

```
Before sorting:
Numbers:   23    6   17   35   33   15   26   12
Letters:    B    H    C    D    A    R    Z    O
After sorting:
Numbers:    6   12   15   17   23   26   33   35
Letters:    A    B    C    D    H    O    R    Z
```

Merge Sort

The Bubble Sort algorithm above is much too slow for the project we have in mind: its worse-case complexity is $\mathcal{O}(N^2)$ for a list of length N.

Instead we'll implement the **Merge Sort** algorithm, which you may remember from CS209. It has complexity $\mathcal{O}(N \log N)$.

Merge Sort

- Split the list into two smaller lists,
- Split each of those into 2 smaller lists.
- Keep doing this until each list is of length 1.
- A list of length 1 is already sorted, so...
- Reassemble each of your sub-lists by merging these sorted list.

Merge Sort

It is useful to write this as a **recursive algorithm**:

Recursive Merge Sort Algorithm

procedure $mergesort(L = a_1, a_2, \ldots, a_n)$
if $n > 1$ *then*

$m := \text{floor}(n/2)$
$L_1 := (a_1, a_2, \ldots, a_m)$
$L_2 := (a_{m+1}, a_{m+1}, \ldots, a_n)$
$L := merge\big(mergesort(L_1), mergesort(L_2)\big).$

end if

So we need two functions:

(i) A `Merge()` function to merge two sorted list
(ii) A `MergeSort()` function that
- splits the list in two,
- calls `MergeSort()` for each half
- calls the `Merge()` function

Merge Sort

Why is Merge Sort faster than Bubble?

Our first function will take two sorted lists, and combine them.

02MergeSort.cpp

```
   template <typename MyType>
56 void Merge(MyType *list1, unsigned int length1,
             MyType *list2, unsigned int length2,
58           MyType *Merged)
   {
60   unsigned int i=0, j=0;
     for (unsigned int k=0; k<length1+length2; k++)
62     if ( (i != length1) && ((j==length2)
                               || (list1[i] <= list2[j])) )
64     {
         Merged[k] = list1[i];
66       i++;
       }
68     else
       {
70       Merged[k] = list2[j];
         j++;
72     }
   }
```

02MergeSort.cpp

```
template <typename MyType>
void MergeSort(MyType *list, unsigned int length)
{
  if (length <=1) // A list of length 0 or 1 is sorted.
    return;
  else
  {
    unsigned int m;
    m = (unsigned int)floor((double)length/2.0);
    MyType *list1 = new MyType [m];
    MyType *list2 = new MyType [length-m];
    for (unsigned int i=0; i<m; i++)
      list1[i]=list[i];
    for (unsigned int i=0; i<length-m; i++)
      list2[i]=list[m+i];
    MergeSort(list1, m);
    MergeSort(list2, length-m);
    Merge(list1, m, list2, length-m, list);
    delete [] list1;
    delete [] list2;
  }
}
```

Analysing the passwords file

Now that we can sort the data, we can work on the password file.

The first step is to open the file, and count the number of lines, and the length of the longest line.

03SortPasswords.cpp

```
   std::ifstream InFile;
36 std::string InFileName="UserAccount-1e4.txt";

38 InFile.open(InFileName.c_str());
   if (InFile.fail() )
40 {
     std::cerr << "Error: Cannot open " << InFileName <<
42     " for reading." << std::std::endl;
     exit(1);
44 }

46 // Need to know the number of lines, and the length of the longest one
   unsigned int LineCount=0, LongestLine;
48 LineCount =  FileLength(InFile, LongestLine);
```

Analysing the passwords file

03SortPasswords.cpp

```
116 int FileLength(std::ifstream &InFile, unsigned int &LongestLine)
    {
118   InFile.clear();
      InFile.seekg(std::ios::beg); // Rewind to the start of the file

      char c;
122   InFile.get( c );
      unsigned int LineCount=0, ThisLineLength=0;
124   LongestLine=0;
      while( ! InFile.eof() )
126   {
        if (c != '\n')
128       ThisLineLength++;
        else
130     {
          LineCount++;
132       if (LongestLine<ThisLineLength)
            LongestLine = ThisLineLength;
134       ThisLineLength=0;
        }
136     InFile.get( c );
      }
138   InFile.clear();
      InFile.seekg(std::ios::beg);     // Rewind
140   return(LineCount);
    }
```

Analysing the passwords file

Now read the file (again) and store the passwords in an array. Again, we write a single stand-alone function to do this.

03SortPasswords.cpp

```
150 void ReadPasswords(std::ifstream &InFile, std::string *Passwords,
                     unsigned int &LineCount, unsigned int LongestLine)
152 {
      int WordsRead=0;
154   char *c_string_word = new char [LongestLine+1];
      for (unsigned int Line=0; Line < LineCount; Line++)
156   {
        InFile.getline(c_string_word, LongestLine+1);
158     Passwords[Line] = c_string_word;
        if (Passwords[Line].length() == 0) // that was a blank line
160       Line--;
        else
162       WordsRead++;
      }
164   LineCount = WordsRead;
      delete [] c_string_word;
166 }
```

Analysing the passwords file

The next step (main, Line 53) is to call the `MergeSort()` function. We then have the task of finding which word occurs most frequently. The approach is to create to new arrays:

(a) a new list of `strings`, called `UniqueWords`, where each password appears, but only once.

(b) a corresponding `int` array `WordFreq`. When we are done, if `WordFreq[k]=x`, then `UniqueWords[k]` appeared x times in the original list.

`03SortPasswords.cpp`

```
   std::string *UniqueWords = new std::string [LineCount+1];
66 unsigned int *WordFreq  = new unsigned int [LineCount+1];
   unsigned int UniqueWordsFound;

// The first one can't already be on the list
70 UniqueWords[0] = Passwords[0];
   WordFreq[0] = 1;
72 UniqueWordsFound=1;
```

continued...

Analysing the passwords file

03SortPasswords.cpp

```
74    for (unsigned int i=0; i<LineCount; i++)
      {
76      if (Passwords[i] !=  UniqueWords[UniqueWordsFound-1])
        { // We have found a new word
78        UniqueWords[UniqueWordsFound] = Passwords[i];
          WordFreq[UniqueWordsFound] = 1;
80        UniqueWordsFound++;
        }
82      else
          WordFreq[UniqueWordsFound-1]++;
84    }
```

Explanation:

Analysing the passwords file

Our next step is to create a list to the 10 most frequently used. This information will be stored in two arrays:

```
string Top10[10];
int Top10Freq[10];
```

We will keep this list ordered. Then iterate through the `UniqueWords` list. If we find a word that occurs more often than the (current) 10th most common, we insert it into the list:

03SortPasswords.cpp

```
   // Insert the 1st into list and set rest to 0
90 Top10[0]=UniqueWords[0];
   Top10Freq[0]=WordFreq[0];
92 for (int i=1; i<10; i++)
   {
94   Top10[i]="";
     Top10Freq[i]=0;
96 }
// See if this is at least as freq as the 10th most
98 for (unsigned int i=1; i<UniqueWordsFound; i++)
     if (WordFreq[i] > Top10Freq[9])
100    Insert(Top10, Top10Freq, UniqueWords[i], WordFreq[i]);
```

Analysing the passwords file

To finish, we'll see how the `Insert` function works:

03SortPasswords.cpp

```
// Insert NewString into the list Top10, ordered by
// NewCount in Top10Freq, bumping anything if needed
void  Insert(std::string *Top10, unsigned int *Top10Freq,
             std::string NewString, unsigned int NewCount)
{
  if (NewCount <= Top10Freq[9])
    std::cerr << "Error: new entry would not make top 10" << std::std::endl
  else
  {
    Top10[9]=NewString;
    Top10Freq[9]=NewCount;
    for (int i=8; i>=0; i--)
    {
      if (Top10Freq[i]<NewCount)
      {
        Top10[i+1] = Top10[i];
        Top10Freq[i+1] = Top10Freq[i];
        Top10[i]=NewString;
        Top10Freq[i]=NewCount;
      }
    }
  }
}
```

Sorting: A note on complexity

Today, we considered two sorting algorithms

- **Bubble Sort** which is conceptually simple, and has a worst-case complexity of $\mathcal{O}(N^2)$ for a list of length N.
- A recursive **Merge Sort**, which has a worst-case complexity of $\mathcal{O}(N \log N)$ for a list of length N.

This means that if we have a list of length N, then the expected time taken for the methods are $C_B N^2$ and $C_M N \log N$, for some constants C_B and C_M. Why? See notes on board...

Sorting: A note on complexity

We want to estimate these constants so that we can predict how long the algorithm will take for some given N.

Before class, I ran both algorithms. Here is a snippet of the code I used, and the output. **Can we estimate how long each algorithm would take for a list of length 32 million?**

Sorting: A note on complexity

See `04CompareSorts.cpp` for more details

```
70    for (int n=1000; n<=32000; n*=2) {
        for (int i=0; i<n; i++)
72        Copy[i] = Passwords[i];

74      start=clock();
        BubbleSort(Copy,n);
76      diff = (double)(clock()-start);
        diff_seconds = diff/CLOCKS_PER_SEC;
78      C_bubble = diff_seconds/( (double)(n*n));
        std::cout << "Bubble: n = " << std::setw(7) << n << " took "
80                << diff_seconds << std::setw(10) << " seconds. C="
                  << C_bubble << std::endl;
82    }
```

Output (Bubble Sort):

```
Bubble: n =  1000 took  0.02 seconds. C=2e-08
Bubble: n =  2000 took  0.11 seconds. C=2.75e-08
Bubble: n =  4000 took  0.44 seconds. C=2.75e-08
Bubble: n =  8000 took  1.76 seconds. C=2.75e-08
Bubble: n = 16000 took  7.05 seconds. C=2.75e-08
Bubble: n = 32000 took 28.36 seconds. C=2.76e-08
```

Sorting: A note on complexity

See 04CompareSorts.cpp for more details

```
98    for (int n=50000; n<=32*50000; n*=2) {
        for (int i=0; i<n; i++)
100       Copy[i] = Passwords[i];
        start=clock();
102     MergeSort(Copy,n);
        diff = (double)(clock()-start);
104     diff_seconds = diff/CLOCKS_PER_SEC;
        C_merge= diff_seconds/( (double)(n*log2(n)));
106     std::cout << "Merge: n = " << std::setw(7) << n << " took "
                  << diff_seconds << std::setw(10) << " seconds. C="
108               << C_merge << std::endl;
      }
```

Output (Merge Sort):

```
Merge: n =    50000 took 0.07 seconds. C=1.293e-07
Merge: n =   100000 took 0.13 seconds. C=1.129e-07
Merge: n =   200000 took 0.3  seconds. C=1.228e-07
Merge: n =   400000 took 0.69 seconds. C=1.337e-07
Merge: n =   800000 took 1.58 seconds. C=1.453e-07
Merge: n =  1600000 took 3.69 seconds. C=1.614e-07
```