CS319: Scientific Computing (with C++)

The Standard Template Library (and sparse matrix formats)

Week 10: **9am** and **4pm**, 15 March 2017

Recall: Sparse Matrices

Last week we

- studied an algorithm for analyzing network data;
- introduced the idea of sparse matrices, and their formats.

Recall: a matrix is `sparse` if the **number of non-zeros** (**NNZs**) is relatively small compared to the overall size of the matrix. To exploit this, we want to store only the non-zeros. Formats for doing this include

- **Triplet** (which we'll looked at last week),
- **Compressed Row Storage** (CRS) (after triplet)
- **Compressed Column Storage** (CCS) (very similar to CRS)
  And the following formats for very specialised matrices (we won't cover them):
- **Block Compressed Row/Column Storage**
- **Compressed Diagonal Storage**
- **Skyline**

Recall: Triplet Format

The basic idea for triplet form is:

Example: write down the triplet form of the following matrix:

$$\begin{pmatrix} 1 & 10 & 11 & 0 & 0 \\ 1 & 0 & 2 & 0 & 3 \\ 9 & 19 & 0 & 29 & 0 \\ 4 & 0 & -4 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \end{pmatrix}$$

Recall: Triplet Format

Towards the end of the last class, we looked at an implementation for `triplet`, including methods for setting and getting values, and for matrix-vector multiplication.

`Triplet.h` ← link!

```
1  // Triplet.h:  For CS319 Week 9
   // Author: Niall Madden,
3  //   Date: 08 March 2017

5  #ifndef _TRIPLET_H_INCLUDED
   #define _TRIPLET_H_INCLUDED
7
   #include "Vector09.h"
9  #include "Matrix09.h"

11 class triplet {
     friend triplet full2triplet(matrix &F, unsigned int NNZ_MAX);
13 private:
     unsigned int *I, *J;
15   double *X;
     unsigned int N;
17   unsigned int NNZ;
     unsigned int NNZ_MAX;
```

Recall: Triplet Format

Triplet.h ⟵ link!

```
20  public:
      triplet (unsigned int N, unsigned int nnz_max); // Constructor
22    triplet (const triplet &t); // Copy constructor
      ~triplet(void);
24
      triplet &operator=(const triplet &B); // overload assignment operator
26
      unsigned int size(void) {return (N);};
28    int where(unsigned int i, unsigned int j); // allow negative return
      // on errors
30    unsigned int nnz(void) {return (NNZ);};
      unsigned int nnz_max(void) {return (NNZ_MAX);};
32
      double getij (unsigned int i, unsigned int j);
34    void setij (unsigned int i, unsigned int j, double x);

36    unsigned int getI (unsigned int k) { return I[k];};
      unsigned int getJ (unsigned int k) { return J[k];};
38    double getX (unsigned int k) { return X[k];};

40    vector operator*(vector u);
      void print(void);
42  };
```

Recall: Triplet Format

We went (quickly) through the constructor for the `triplet` class last week, and the `setij` function. The latter depended on the `where` function. Here is the code for that:

`Triplet.cpp` ← link!

```
// triplet::where
// return k if A(i,j) is stored in X[k].
// return -1 if A(i,j) is not stored.
int triplet::where(unsigned int i, unsigned int j)
{
  unsigned int k=0;
  do {
    if ( (I[k]==i) && (J[k]==j) )
      return(k);
    k++;
  } while (k<NNZ);
  return(-1);
}
```

The use of the `Triplet` class is demonstrated in a programme called `01triplet_example`. It shows how to use the **Jacobi** method to solve a linear system where the matrix is stored in `triplet` format.

Compressed Column Storage

If we know that the entries in our matrix are stored in order than it is possible to store the matrix more efficiently that in Triplet format. One way of doing this is to use **Compressed Column Storage**, also known as *Harwell-Boeing*

The matrix is stored in 3 vectors:

- a `double` array, x of length `nnz` ("number of nonzero entries") storing the non-zero entries matrix, in column-wise order.
- an `int` array, r of length `nnz` storing column index of the entries. That is, $x[k]$ would be found in row $r[k]$ of the full matrix.
- an `int` array, c of length $N + 1$, where $c[k]$ stores that starting point of column k as it appears in the arrays x and r, and $c[N] = nnz$.

Example:

$$\begin{pmatrix} 2 & -1 & 0 & 0 & -2 \\ -3 & 5 & -1 & 0 & 0 \\ 0 & -2 & 4 & -2 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ -3 & 0 & 0 & -1 & 4 \end{pmatrix}$$

The process of multiplying a matrix (in CCS) by a vector is rather simple:

```
int ind=0;
for (int col=0; col<N; col++)
   for (j=c[col]; j<c[col+1]; j++)
   {
     i=r[ind];
     v[i] += x[ind]*b[j];
     int ++;
   }
```

I don't provide code for implementing a CCS class here: that is an exercise.

(Not) Reinventing the wheel

During the semester, we've focused on designing classes that can be used to solve problems. These included classes: `stack`. `vector` and `matrix`.

However, most of you worked out that, to some extent, these are already supported in C++. The motivations for reinventing them included

- our implementation is simple to use;
- we learned important aspects of C++/OOP;
- we needed to achieve specific tasks efficiently: this is particularly true of our design of sparse matrix classes.

Now we'll look at how to use the built-in implementation that comes with the C++ **Standard Template Library (STL)**.

The Standard Template Library (STL)

The **STL** provides

(1) **Containers:** ways of collecting/storing items of some type (template....)

(2) **Iterators:** for accessing items in the containers

(3) **Algorithms:** for operating on the contents of containers, such as finding a particular item, or sorting (a subset) of them.

(4) **functors:** essentially, a class which defines the operator(). We won't say more than this right now.

We'll now look at examples of (1)–(3), and then consider an application to our password frequency problem from Week 6.

It has to be noted, though: the STL is not that easy to use. In particular the error messages generated are rather verbose and unhelpful.

A **container** stores objects/elements. These elements can have basic data-type (e.g., `char`, `int`, `double`, ...) or can be objects (e.g., `string`, or user-defined objects).

The most important types of containers are:

- `vector`: an indexed sequence (often called "*random access*", though this would be better called "*arbitrary access*". All the items are of the same type. It can be resized, and have new items added to the end. One can also add items to positions not the end, but this is slow.
- `set`: a collection of unique items (of the same type), stored in order. When defined relative to a user-defined class, an overloaded `operator::operator<` (less than) must be provided for correct operation.
- `multiset`: an ordered collection, like a set, but can have repeated values.
- `list`: a doubly linked list.
- `stack`: a stack.
- ... etc...

In this class, we'll focus on `set`s, `multiset`s and `vector`s.

An `iterator` is an object used to select (or move between) elements in a container.

We can think of them as pointers, that allow us to reference particular elements.

They come in particular flavours:

- forward, reverse, and bidirectional iterators;
- random-access/indexed-access iterators;
- input and output iterators;

sets and multisets

To use a set or multiset, we must

```
#include <set>
```

Suppose we want to create a multiset to store strings (which just happen to be passwords...), and an iterator for it, we could define

```
std::multiset <std::string> multi_pwd;
std::multiset <std::string>::iterator multi_pwd_i;
```

To add an item to the (multi)set, we could used

```
multi_pwd.insert(MyString);
```

This will add the new string to the multiset, automatically choosing its position so that it remains ordered. (If we use a set, it gets inserted into the correct position, providing this does not result in duplication).

sets and multisets

Other important methods include

- `begin()` (returns an iterator that points to the first element)
- `end()` (returns an iterator that points to *one past* the end of the set).
- `clear()` (remove contents)
- `count()` (count number of occuences)
- `empty()` (is the set empty?)
- `erase()` (remove an element, or range of elements)
- `find()` (locate an element; return an iterator)
- `size()` (number of elements)
- `swap()` (swap contents of two sets of same type)
- `for_each()` (apply a particular function to each item in a container)

An example of using `begin` and `end` with a set and multiset:

02set_and_multiset.cpp

```
12  int main(void )
    {
14    std::set <int> set_int;
      std::set <int>::iterator set_int_i;
16    std::multiset <int> multi_int;
      std::multiset <int>::iterator multi_int_i;
18
      for (int i=0; i<=20; i+=3) // (0,3,6,9,12,15,18)
20    {
        set_int.insert(i);
22      multi_int.insert(i);
      }
24    for (int i=20; i>0; i-=2) // (20,18,16,...,4,2)
      {
26      set_int.insert(i);
        multi_int.insert(i);
28    }
```

Now we see how to iterate over the multiset:

02set_and_multiset.cpp

```
30    std::cout << "The multiset has " << multi_int.size() <<
        " items." << std::endl;
32    std::cout << "\t They are: ";
      for (multi_int_i = multi_int.begin();
34         multi_int_i != multi_int.end();
           multi_int_i++)
36      std::cout << std::setw(3) << *multi_int_i;
      std::cout << std::endl;
38    std::cout << "\t 6 occurs " << multi_int.count(6) <<
        " time(s)." << std::endl;
```

The output is

```
1  The multiset has 17 items.
      They are:   0  2  3  4  6  6  8  9 10 12 12 14 15 16 18 18 20
3     6 occurs 2 times.
```

And then over the set:

02set_and_multiset.cpp

```
  std::cout << "The set has " << set_int.size() <<
42  " items." << std::endl;
  std::cout << "\t They are: ";
44  for (set_int_i = set_int.begin();
       set_int_i != set_int.end();
46     set_int_i++)
    std::cout << std::setw(3) << *set_int_i;
48  std::cout << std::endl << "\t 6 occurs " << set_int.count(6) <<
    " time(s)." << std::endl;
```

The output is

```
1 The set has 14 items.
         They are:   0  2  3  4  6  8  9 10 12 14 15 16 18 20
3        6 occurs 1 time.
```

vector

To use `vector`, we must

```
#include <vector>
```

Unlike a set, we can access a vector by index. Moreover, by default it is not sorted, though there are algorithms to sort its contents....

Since it is unordered, a new item usually gets added to the end, using `push_back`

This can be removed, using `pop_back`

Other important methods include

- `at`
- `operator[]`
- `back` (not the same as `end`)
- etc.

03STL_vector.cpp

```
#include <vector>        // vector
#include <algorithm>     // sort

void print_int (int i) { std::cout << std::setw(3) << i; }

int main(void )
{
  std::vector <int> vec_int;
  std::vector <int>::iterator vec_int_i;

  std::cout << "Vector has " << vec_int.size() <<
    " elements." << std::endl ;

  for (int i=3; i>=0; i--)
    vec_int.push_back(i*3); // (9,6,3,0)

  std::cout << "Vector has " << vec_int.size() << " elements: ";
  for (unsigned int i=0; i<vec_int.size(); i++)
    std::cout << std::setw(3) << vec_int[i];
```

Output (so far):

```
Vector has 0 elements.
Vector has 4 elements:   9   6   3   0
```

This snippet demonstrates the use of

- the `find` and `insert` methods;
- the `for_each` iterate through an entire container.

03STL_vector.cpp

```
  vec_int_i = find (vec_int.begin(),vec_int.end(),3);
32
  vec_int.insert(vec_int_i,10);
34
  std::cout << std::endl;
36 std::cout << "Vector has " << vec_int.size() << " elements: ";
  for_each (vec_int.begin(), vec_int.end(), print_int);
```

Output (continued):

```
1 Vector has 0 elements.
  Vector has 4 elements:   9  6  3  0
3 Vector has 5 elements:   9  6 10  3  0
```

Finally, we show how to `sort` the items in the list:

03STL_vector.h ← link!

```
40    std::cout << "Sorting the vector..." << std::endl;
      sort(vec_int.begin(), vec_int.end());
42    std::cout << "Now vector is: ";
      for_each (vec_int.begin(), vec_int.end(), print_int);
```

Output (all):

```
1  Vector has 0 elements.
   Vector has 4 elements:   9  6  3  0
3  Vector has 5 elements:   9  6 10  3  0
   Sorting the vector...
5  Now vector is:   0  3  6  9 10
```

Range-based for loops

The **ranged-based** *for* loop is a recent addition to C++ (so it might not work with old compilers. With g++, you may need to enable the `c++11` option.

In the code above, the line

```
1    for_each (vec_int.begin(), vec_int.end(), print_int);
```

could be replaced with

```
1    for (int i : vec_int)
        print_int(i);
```

Algorithm

To use `algorithm`, we must

```
#include <algorithm>
```

Useful functions that this provides include

- `for_each`
- `sort` and `partial_sort`
- `search`
- `copy` and `fill`
- `merge`
- `set_union`, `set_difference`
- etc.

The password frequency problem

(See lecture notes for details)

```
#include <set>        // multiset
#include <vector>     // vector
#include <algorithm> // sort
```

The password frequency problem

```
1  class pwd {
   private:
3    std::string word;
     int freq;
5  public:
     pwd(std::string s, int f) {word=s; freq=f;};
7    std::string getword(void) const {return(word);};
     int getfreq(void) const {return(freq);};
9  };

11 bool comp (pwd p, pwd q)
   {
13   return (p.getfreq() > q.getfreq());
   }
15
   int FileLength(std::ifstream &InFile, int &LongestWord);
```

The password frequency problem

```
int main(void )
{
  std::ifstream InFile;
  std::string InFileName="UserAccount-1e5.txt";
  std::multiset <std::string> multi_pwd;
  std::multiset <std::string>::iterator multi_pwd_i;
  std::vector <pwd> vector_pwd;

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

  // Need to know the number of lines, and the length of the longest one
  int LineCount=0, LongestLine;
  LineCount =  FileLength(InFile, LongestLine);
  std::cout  << InFileName << " has " << LineCount << " lines.\n";
  std::cout  << "\tThe longest has " << LongestLine << " characters.\n";
```

The password frequency problem

```
// Read the lines
char *c_string_word;
c_string_word = new char [LongestLine+1];
for (int i=0; i<LineCount; i++)
{
  InFile.getline(c_string_word, LongestLine+1);
  multi_pwd.insert(c_string_word);
}
```

The password frequency problem

```
// Copy the passwords to the pwd vector
multi_pwd_i = multi_pwd.begin();
vector_pwd.push_back(pwd(*multi_pwd_i,
      multi_pwd.count(*multi_pwd_i)));
multi_pwd_i++;
while (multi_pwd_i != multi_pwd.end())
{
  if ((vector_pwd.back()).getword() != *multi_pwd_i)
    vector_pwd.push_back(pwd(*multi_pwd_i,
       multi_pwd.count(*multi_pwd_i)));
  multi_pwd_i++;
}
```

The password frequency problem

```
  std::sort (vector_pwd.begin(), vector_pwd.end(), comp);

  std::cout << "Top 10 passwords are: " << std::endl;
  for (unsigned int i=0; i<10; i++)
    std::cout <<  std::setw(12) << (vector_pwd[i]).getword() <<
      std::setw(6) << (vector_pwd[i]).getfreq() << std::endl;

  InFile.close();

  return(0);
}
```