Modern C++ - STL example

19. An STL example

In this chapter we discuss the practical usage of STL. We will solve a simple task without STL and then we will introduce STL instrments step by step.

Merging two sorted file

Suppose we have two sorted files of strings *file1* and *file2*. The task is to create *file3* as a sorted output where we merge the values from the input files.

At the beginning we create a solution without using STL at all.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

// simple merge
int main()
{
  string s1, s2;
  ifstream f1("file1.txt");
  ifstream f2("file2.txt");
  ofstream f3("file3.txt");
  f1 >> s1;
  f2 >> s2;
  // the usual way:
  while (f1 || f2)
  {
    if (f1 && ((s1 <= s2) || !f2))
    {
      f3 << s1 << endl;
      f1 >> s1;
    }
    if (f2 && ((s1 >= s2) || !f1))
    {
```

```
26       f3 << s2 << endl;
27       f2 >> s2;
28     }
29   }
30   return 0;
31 }
```

In lines 11-13 we open the files. In lines 14-15 we read one-one elements from both input streams. This will indicate *eof* state if any of the files is empty.

We are looping until *all* input files are exhausted. To reach this, in line 19 and 24 we compare the two pre-read elements, and we select the smaller one. The trick in the comparision is that we also select the element if the other file is already empty. the selected element is written to the output and a new pivot element is read from the same file.

Although this solution is working fine and effectively, it is easy to see that its complexity make it hard to maintain. For example, if the client requirement changes to use a different way of comparision, or handle *eof* in a different way, we have to modify the critical part of the loop, and then have to re-test the whole program.

Using STL in a naïve way

In the following iteration we will use teh **std::vector** container and tthe **std::merge** algorithm in a naïve way: we read both input files into two vector of **std::string** and then merge them into a third output vector.

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>    // for merge( b1, e1, b2, e2, b3 [,opc_comp])
#include <vector>

using namespace std;

int main()
{
  ifstream f1("file1.txt");
  ifstream f2("file2.txt");
  ofstream f3("file3.txt");
  string s;
  vector<string> v1;
16   while ( if1 >> s ) v1.push_back(s);
17   vector<string> v2;
18   while ( if2 >> s ) v2.push_back(s);
19
20   // allocate the space for the result
21   vector<string> v3(v1.size()+v2.size());   // expensive...
22
23   merge( v1.begin(), v1.end(),
24          v2.begin(), v2.end(),
25          v3.begin());              // v3[i++] = *current++
26   for ( int i = 0; i < v3.size(); ++i)
27     f3 << v3[i] << endl;
28   return 0;
29 }
```

We read the content of input files into **v1** and **v2** in lines 16 and 18 respectively. As **merge** put the output into a fully pre-allocated container we have to create **v3** with the necessary size in line 21. This is an expensive operation as we have to create an empty string into every vector element – just to be overwritten in the next statement.

The actual merge happens in line 23-25 in the **merge** function, where the first four parameters define the input intervals, and the fifth parameter is the iterator to the beginning of the output interval. The merge algorithm supposes that thelength of the output interval is long enough, that we fulfilled in line 21.

Although, the program itself is trivial now, it has both functional and performance problems. First of all: the input file sizes are limited by the memory of the program: in one point we keep all input strings twice in the memory: (ones as the input either in **v1** or **v2** and as the output in **v3**). The unnecessary creation of empty strings in line 21 is also a burden.

Using insert adaptors

Although, it does not really changes the major problems with the program above, we can solve the last issue. It would be a logical choice not to preallocate (and then owerwrite) all output elements, but extend the vector using the **push_back** method one by one.

First of all, we do not allocate all elements of the output vector in advance, we just create an empty vector **v3** in line 22. Since extending a vector with N elements at the end reallocates the buffer of a vector roughly *log(N)* times, and copies the elements, we pre-allocatethe buffer with the **reserve** method in the next line in one step. After calling **reserve** the *logical* size of **v3** is still zero, only the *physical buffer* is allocated.

The issue comes with the **merge** function. Merge wants to *overwrite* the output vector and not to *extend* it. Here we apply an *adaptor*.

```
 1 #include <iostream>
 2 #include <fstream>
 3 #include <string>
 4 #include <algorithm>    // for merge( b1, e1, b2, e2, b3 [,opc_comp])
 5 #include <vector>
 6 #include <iterator>     // for back_insert_iterator
 7
 8 using namespace std;
 9
10 int main()
11 {
12   ifstream f1("file1.txt");
13   ifstream f2("file2.txt");
14   ofstream f3("file3.txt");
15   string s;
16   vector<string> v1;
17   while ( f1 >> s ) v1.push_back(s);
18   vector<string> v2;
19   while ( f2 >> s ) v2.push_back(s);
20
21   // do not allocate the space for the result
22   vector<string> v3;                // cheap...
23   v3.reserve(v1.size()+v2.size()); // allocate all space in once
24
25   merge( v1.begin(), v1.end(),
26          v2.begin(), v2.end(),
27          back_inserter(v3));       // v3.push_back(*current++)
28   for ( int i = 0; i < v3.size(); ++i)
29     f3 << v3[i] << endl;
30   return 0;
31 }
```

Adaptors modify the interface of a container (or an iterator). That is here we will replace **v3.begin()** with a **back_inserter_iterator** object in line 27. The back_insert_iterator class is defined in header <iterator> included in line 6 and the object itself is created by the **back_inserter** factory function.

A back_inserter_iterator adaptor is *acting* like an output iterator, but its **operator=**

instead of overwriting the referred element on the container calls the **push_back** method on the original container (here on **v3**). Thus merge *thinks* that overwrites the output range, but in reality it *extends* it.

There are also inserters to insert elements into front calling **push_front** or just **insert** for associative containers.

Input and output iterator adaptors

The major issue with the previous program version was to store all elements in the memory (twice) in temporary vectors.

To solve this we will apply adaptors for the input and output files. Such adaptors are called *input* and *output iterator adaptors*.

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <iterator>     // input- and output-iterators

using namespace std;

int main()
{
  ifstream f1("file1.txt");
  ifstream f2("file2.txt");
  ofstream f3("file3.txt");
  // istream_iterator(if1) -> f1 >> *current
  // istream_iterator() -> EOF
  // ostream_iterator(f3,x) -> of << *current << x
  merge( istream_iterator<string>(f1), istream_iterator<string>(),
         istream_iterator<string>(f2), istream_iterator<string>(),
         ostream_iterator<string>(f3,"\n") );
  return 0;
}
```

The iterator adaptors have the same idea as the inserters. They are acting like iterators but reading from and write to the wrapped stream objects. To express the *end* of the input range (here the *eof* event on read), default constructed istream_iterator objects are given as the first and third parameter of merge.

As the same stream can be read as sequence of characters, numbers of strings, the **input_iterator** class is a template to define the unit to read (here string). For output iterator we can define a *sepearator* to be written after every output action (here **"\n"**).

Notice that in this version there is no vectors to buffer input and output, therefore our program has a low memory footprint and can also work on infinite input streams.

Functors

Suppose, later the customer of our program wants a change: in the sorting of the strings we have to negligate the difference between upper case and lower case letters. The problem is, that the **merge** algorithm by default uses the **operator<** between the input elements, here **string** objects.

In fact, merge has a second version, where the sixth parameter is used to compare the input elements. Such *functors* are classes acting like dunctions, defining the *function call operator* (**operator()**). Functors can appear in various ways, but *unary predicates*, taking one parameter and returning **bool**, and *binary predicates* taking two parameters and returning **bool** has a major role for STD algorithms. Unary predicates are used in algorithms, like **find_if** or **remove_if**, while binary ones mainly as comparisions.

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <iterator>

using namespace std;

struct my_less // function object: "functor"
{
  bool operator()(const string& s1, const string& s2) const
  {
     string us1 = s1;
     string us2 = s2;
     // TODO: use locale object
     transform( s1.begin(), s1.end(), us1.begin(), toupper);
     transform( s2.begin(), s2.end(), us2.begin(), toupper);
     return us1 < us2;
   }
```

```cpp
};
int main()
{
  ifstream f1("file1.txt");
  ifstream f2("file2.txt");
  ifstream f3("file3.txt");

  merge( istream_iterator<string>(f1), istream_iterator<string>(),
         istream_iterator<string>(f2), istream_iterator<string>(),
         ostream_iterator<string>(f3,"\n"), my_less() );
  return 0;
}
```

The **my_less** class has a function call operator method to compare two strings and returning **true** when the case insensitive comparison recognizes that the first parameter is *less* than the second. This class is *callable*. Thus it acts as a *binary predicates*.

A temporary object is created as the sixth parameter of merge algorithm and its function call operator is called for every time merge compares two strings.

The implementation of the function call operator is only demonstration purposes, a real solution would utilize the **local** library.

Functor with state

More complex functors can be defined using the full feature set of object-oriented programming.

For example, the customer may not simply want to merge the input files in a sorted way, but taking one element from file1 and than one from file2, “zipping” the files. We can modify the previous solution to returning the predicates alternating **true** and **false**. This requires to strore the previous state: we will create an attribute in the functor for this purpose.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <iterator>

```

```cpp
using namespace std;

struct zipper
{
  zipper() : _flag(false) {}
  bool operator()(const string&, const string&) const
  {
    _flag = ! _flag;
    return _flag;
  }
  bool _flag;
};
int main()
{
  ifstream f1("file1.txt");
  ifstream f2("file2.txt");
  ifstream f3("file3.txt");

  merge( istream_iterator<string>(f1), istream_iterator<string>(),
         istream_iterator<string>(f2), istream_iterator<string>(),
         ostream_iterator<string>(f3,"\n"), zipper() );
  return 0;
}
```

Usually, it is not suggested to define functors with *state*. Some STL algorithm may copy or assign the functor and therefore its behavior may hard to predict. In such simple cases, however, we can use predicates with state.

Notice, that since the function call operator does not depend on its parameters, we declare them without formal parameter name.

Generalization

In the last step we generalize the zipper class. We make it template, so it can work for any type we can read in and write out to files. (We do not use any comparision anymore, so this is not a restriction now.)

We can apply further (run-time) parameters to tell teh zipper how many elements to read from “left” (file1) and from “right” (file2). Also we can specify whether to start from left or right.

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <algorithm>
#include <iterator>

using namespace std;

template <typename T>
class zipper
{
public:
  zipper(int l, int r, bool fl = true) :
        left(l), right(r), from_left(fl), cnt(0) { }
  bool operator()( const T&, const T&) const
  {
    bool ret = from_left;
    const int  max = from_left ? left : right;
    if ( ++cnt == max )
    {
      cnt = 0;
      from_left = ! from_left;
    }
    return ret;
  }
private:
  const int left;
  const int right;
  const bool from_left;
  int cnt;
};
int main()
{
  ifstream f1("file1.txt");
  ifstream f2("file2.txt");
  ifstream f3("file3.txt");

  merge( istream_iterator<string>(f1), istream_iterator<string>(),
            istream_iterator<string>(f2), istream_iterator<string>(),
            ostream_iterator<string>(f3,"\n"), zipper() );
  return 0;
}
```

A further advantage of such STL-style programming is that we separated the functor logic: a different individual can code (and test) it, while somebody else can write the other part of the program.