Tutorial 3 – Template, String, Streams, Vector, Iterator

(Week 5, starting 5 September 2016)

1. *Template Class, New Data Structures*

You are given a template Pair<TL, TR> class. Each object of this class can point to 2 objects of different types:

Related Concepts

- Template
- Pass-by-value vs pass-by-pointer
- Operators: `.` vs `->`

```
template <typename TL, typename TR>
class Pair {
     TL* _objLeft; TR* _objRight;
  public:
     Pair(TL* pobjLeft, TR* pobjRight) :
          _objLeft(pobjLeft), _objRight(pobjRight) {}
     TL* getLeft() { return _objLeft; }
     TR* getRight() { return _objRight; }
};
```

We now want to create a TemplateTriple<TL, TM, TR> class. Each object of this class can **point to 3 objects** of different types. **Restriction** for each of parts (a) - (d): Your class should have only **ONE member variable**, and you should be using the Pair class where possible.

There are 2 different ways to achieve this:

(a) Use inheritance.
(b) Use composition. TemplateTriple is composed of a Pair, i.e. it has a Pair object as a member variable. [Hint: How do you point to 3 objects in a Pair? Use 2 Pair objects, but only as a member variable]

We can now instantiate a TemplateTriple object that points to 3 objects. A Person has a name (string), weight (double), and height (double). Create a Person data structure and use a TemplateTriple to help you store data:

(c) Use inheritance. [Hint: Are you inheriting a family of classes, or just one specific type?]
(d) Use composition. Person is composed of a TemplateTriple.

A **Person** object should have 3 getters, one for each attribute. Each **getter** should return the **value** of the name/weight/height itself, and NOT a pointer to the value.

Answer

Note the difference between the:

- Pair<TL, TR> class given here — stores two pointers to two elements of different types
- Pair<T> class in lectures — stores two elements of the same type
- std::pair<T1, T2> class in <utility> — stores two elements of different types

Aside from this question, we often use the C++ **library data structures** instead of reinventing the wheel:
std::pair in <utility>: http://www.cplusplus.com/reference/utility/pair

```
template <typename TL, typename TM, typename TR>
class TemplateTripleInh : public Pair <TL, TR> { // (a)
     TM* _objMid; // extending left and right is cleanest
  public:        // otherwise, inherited getters will return wrong data
     TemplateTripleInh(TL* pobjLeft, TM* pobjMid, TR* pobjRight) :
          Pair<TL, TR>(pobjLeft, pobjRight), _objMid(pobjMid) {}

     TM* getMid() { return _objMid; }
};

template <typename TL, typename TM, typename TR>
class TemplateTripleComp { // (b)
     Pair<TL, Pair<TM,TR> > _objPair; // object, not pointer!
  public:
     TemplateTripleComp(TL* pobjLeft, TM* pobjMid, TR* pobjRight)
          : _objPair(pobjLeft, new Pair<TM, TR>(pobjMid, pobjRight)) {}
     ~TemplateTripleComp() { delete _objPair.getRight(); }

     TL* getLeft() { return _objPair.getLeft(); }
     TM* getMid() { return _objPair.getRight()->getLeft(); }
     TR* getRight() { return _objPair.getRight()->getRight(); }
}; // watch the . vs -> operator
```

`TemplateTriple<TL, TM, TR>` and `Person` can be viewed as abstract data types, or as data structures with implementation. If we view them as abstract data types, then we know what each of them can do but disregard their internal implementation.

```
class PersonInh : public TemplateTriple <string, double, double> {// (c)
  public:
     PersonInh(string pstrName, double pdblWt, double pdblHt) :
          TemplateTriple(new string(pstrName),
               new double(pdblWt), new double(pdblHt)) {}
     ~PersonInh() {
          delete getLeft(); delete getMid(); delete getRight();
     }

     string getName() { return *getLeft(); } // return object, not ptr!
     double getWt() { return *getMid(); }
     double getHt() { return *getRight(); }
}; // now we have 3 inherited getter methods left hanging...

/* extra: to remove the 3 inherited getter methods,
     we could use private inheritance,
     but this can cause other design problems
*/
```

```
class PersonComp { // (d)
     TemplateTriple<string, double, double> _objTriple;
  public:
     PersonComp(string pstrName, double pdblWt, double pdblHt) :
          _objTriple(new string(pstrName), // why all the new?
               new double(pdblWt), new double(pdblHt)) {}
     ~PersonComp() {
          delete _objTriple.getLeft();
          delete _objTriple.getMid();
          delete _objTriple.getRight();
     }

     string getName() { return *_objTriple.getLeft(); }
     double getWt() { return *_objTriple.getMid(); }
     double getHt() { return *_objTriple.getRight(); }
};
```

2. STL Vector and Iterator

A logistics company uses RFID tags to track the movement of hundreds of thousands of pallets. As pallets arrive, they pass through a scanner, and the pallet ID is added to the end of an STL `vector<string>` called `pallets`.

e.g. pallets [“20-0314”, “20-A921”, “20-A921”, “20-A921”, “20-A921”, “01-0003”, ”D9-3210” …]

Quite often, the same pallet is read repeatedly and consecutively, due to incorrectly configured hardware. We need to remove all consecutive (side-by-side) repeated pallet IDs from the vector `pallets`.

```
void cleanUp(vector<string>& pallets) { // why the & ?
     /* your code here */
}
```

(a) Use a single loop over `pallets`, directly removing the undesired elements one at a time

(b) Do the same as (a), this time using ONLY STL iterators instead of indexes

(c) Can you see that the algorithm in (a) & (b) is inefficient, even though there is just one loop? How do we improve?

Related Concepts

- vector element access - at(), []
- vector erase() & insert() generally inefficient
- iterator ++ -- *, random access
- iterator invalidation

Answer

(a)

```
void cleanUp(vector<string>& pallets) {
     int idx = 1;
     while (idx < pallets.size()) { // while within bounds
          if (pallets.at(idx - 1) == pallets.at(idx))
               pallets.erase(pallets.begin() + idx); // iterator created
          else
               idx++;
     }
}
```

The array subscript operator `pallets[idx]` also works in place of vector's `at(idx)` member function.

(b)

```
void cleanUp(vector<string>& pallets) {
     if (pallets.size() < 2) return;
     vector<string>::iterator prevItr = pallets.begin(), // idx 0
          currItr = pallets.begin() + 1; // idx 1
     while (currItr < pallets.end()) { // while within bounds
          if (*prevItr == *currItr) { // don't forget to dereference
               currItr = pallets.erase(currItr); // currItr invalidated
          } else {
               prevItr++;
               currItr++;
          }
     }
}
```

Removing an element using erase requires left-shifting, invalidating iterators from that index (incl.) onward.

(c)

Each time we remove an element from the vector using `erase(iterator)`, many elements are accessed. Given a large vector of size n, in which every two consecutive elements are repeated, the $\frac{n}{2}$ calls to erase() will, in total, result in **close to** $\frac{n^2}{4}$ elements being shifted.

We can avoid this by iterating through the `pallets` once and **copying** the desired elements out to another vector `buffer`, then **quickly replacing** `pallets`' internal array with `buffer`'s. This requires **proportional to n elements** being accessed / shifted.

```
void cleanUp(vector<string>& pallets) {
     if (pallets.size() < 2) return;
     vector<string> buffer;
     buffer.push_back(pallets.front());
     for (vector<string>::iterator itr = pallets.begin() + 1;
          itr < pallets.end(); itr++)
          if (buffer.back() != *itr)
               buffer.push_back(*itr);
     pallets.swap(buffer);
}
```

Again, we can just use C++ **library functions** instead of reinventing the wheel:

unique() in <algorithm>: http://www.cplusplus.com/reference/algorithm/unique

```
void cleanUp(vector<string>& pallets) { // as efficient as using iter.
 pallets.resize(unique(pallets.begin, pallets.end()) - pallets.begin());
}
```

Wow, just one statement! In one pass through `pallets`, `unique()` removes all adjacent elements that have the same pallet ID, and returns an iterator denoting where the new "end()" should be. As the function does not decrease the size of the container, we need to resize() `pallets` as there may now be less elements within it.

Don't forget that end() is an iterator AFTER the last element, a.k.a. "past-the-end" element.

Note: We say that the algorithm in (a) and (b) is inefficient because it runs in $O(n^2)$ or quadratic time, while (c) runs in O(n) or linear time. The use of iterators may seem meaningless to you now, as array-based lists have random access. However, iterators become very useful for other data structures such as linked lists and hash tables. All these will be learnt later.

3. *String, Streams*

You are interested in finding out the volume and weight of some products. Each product record contains (**product ID**, ☺garbage☺, **volume** in mm^3, **weight** in grams) in that order.

The following are examples of records, all valid:

- **1234567**:Wheel bearing|Yamaha XJ900s|Front:**9000 50**
- 00**900**#acm327df2mm3d1f0#Carburetor needle;Honda CB400;4 pcs;**8 5**
- 000000,Oil filter,Yamaha,3FV-13440-00,**225000 200**

As the data comes from various sources, the delimiter between various parts of the data may be any one of {',', ':', ';', '|', '#'}. The **product ID** is guaranteed to be a non-negative integer, while the (**volume weight**) part is guaranteed to be the only data after the last delimiter.

The above 3 records should be **formatted** as:

```
| 1234567|   9000|  50|
|     900|      8|   5|
|       0| 225000| 200|
```

← Each line is one record

(a) Complete the implementation of the two methods in the given class:

```
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Product {
     long _productID; // any non-negative int is a valid ID
     long _volume; // in cubic mm
     long _weight; // in grams
  public:
     Product(string pInput) { ... } // parse 1 record - set member vars
     string str() { ... } // return the nicely formatted record

     long getProductID() { return _productID; }
     long getVolume() { return _volume; }
     long getWeight() { return _weight; }
};
```

Tip: Check out functions of <string> to help with parsing, that of <iomanip> to help with formatting

(b) Besides returning a formatted string through format(), how can we allow the formatted representation of a Product object to be easily printed?

i.e. How do we enable `cout << someProduct << endl;` to work?

Answer

(a)

Some common string member functions are highlighted:

```
Product(string pInput) { // parse one record
     int firstDelim = pInput.find_first_of(",:;|#");
     int lastDelim = pInput.find_last_of(",:;|#");
     int lastSpace = pInput.rfind(" ");

     string prodID = pInput.substr(0, firstDelim);
     string vol = pInput.substr(lastDelim + 1, lastSpace - lastDelim);
     string wt = pInput.substr(lastSpace + 1, string::npos);

     (istringstream(prodID)) >> _productID;
     (istringstream(vol)) >> _volume;
     (istringstream(wt)) >> _weight;
}
string str() { // return the formatted record
     ostringstream oss;
     oss << "|" << setw(8) << _productID << "|" << setw(7) << _volume
          << "|" << setw(4) << _weight << "|";
     return oss.str();
}
```

npos is a constant member variable of the string class. When used as the length parameter of `substr()`, it indicates "read till the end of string". When `npos` is returned by the ...find...() functions, it indicates "not found".

Reference: http://www.cplusplus.com/reference/string/string/npos/

(b)

We can overload the insertion operator << *outside* of the Product class, so that the formatted representation of a Product object can be fed to an output stream easily.

```
ostream& operator<<(ostream& os, Product& prod) { // outside the class
     os << prod.str();
     return os;
}
```

\- Learn how to learn ☻ -

Explore std library
Test its functions
Code incrementally