Generic Programming

Arun K Rajagopalan, Texas A&M University 10/15/2015

1 Questionable Class Definition

```
template <typename T>
class array {
  T* data;
  int n;
public:
  array (int n_) : data(new T[n_]), n(n_) {}
  ~array () { delete [] data; }

  friend bool operator==(const array<T>& v1, const array<T>& v2) {
    return v1.data == v2.data;
  }
  T& operator[](int i) { return data[i]; }
  int size() { return n; }
};
```

There are many things wrong that make this class definition very bad. Identify those things, describe with an example that manifests the trouble, and suggest a fix. Consider issues, such as passing an array object to a function by value.
To help in this task, consider what pre- and post-conditions and invariants must hold between copy constructor, destructor, equality operator, inequality operator, and the assignment operator. Present these in a tabular form. The goal is to specify what is required from a type to allow its "sensible" use in generic algorithms.
Note, this exercise is not easy. You must first figure out what operations the C++ compiler defines for a class by default, and what are the default definitions of those operations. Then you must consider whether the defaults do the right thing or whether a user-defined implementation is necessary instead.
Ans - The following are some of the things that I feel are wrong with the template definition -

1. **Passing a array object by value** - If we try to do this, the default copy constructor will do a shallow copy, pointing to the same data element, thus causing a double free. The fix is to provide a copy constructor with the desired implementation.

   ```
   void f(array<int> x) {
      return;
   }
   ```

```
int main() {
    array<int> a(10);
    f(a); //Destroyed here
}
```

The fix involves calling a copy constructor on the data member.

2. **Equality operator overloading** - The current implementation just checks the memory address of T* data member. This means that two array object can be equal only if they start at the same location, which does not make sense. A correct implementation would first compare equality on array size and then iterate over all elements, individually checking for element equality. This would then require that the T type be equality comparable.

```
friend bool operator==(const array<T>& v1, const array<T>& v2) {
    if(v1.size() == v2.size()) {
        //Check equality for all members
        return true;
    }
    return false;
  }
```

3. **Copy constructor** - The user would expect a copy constructor to be provided with this class template to copy existing array objects. The default implementation does a shallow copy, which will result in bad behavior. This will also cause a double 'free' condition. The fix will involve defining a custom copy constructor that iterates over the members of data calling their copy constructor (or copy assignment). If this is in-fact forbidden, the function should be deleted.

```
array(const array& a) {
    //call copy constructor on the data object
  }
```

4. **Assignment operator overloading** - Similar to the copy constructor, a user of this template might like to assign existing objects to newly created objects (or existing ones). If this is in-fact forbidden, the function should be deleted.

5. **Safe accesses** - The class does not provide bounds check on array accesses.

6. **Overloaded operator** - Some operations such as >, <, >=, <= can be used to compare two array objects, possibly for use it sorting etc.

2 Tricky STL

STL algorithms can be composed to create new algorithms. Sometimes the amount of code to implement a particular functionality is very small. Here's a fun little example, though perhaps useless:

```
template <typename BidirectionalIterator>
void func1(BidirectionalIterator begin, BidirectionalIterator end) {
  while (std::next_permutation(begin, end));
}
```

- What algorithm does func1 implement?
- Write a program or two that tests its functionality.
- What is the time complexity of the algorithm?

Ans - func1 essentials implements a sorting algorithm. Each iteration of the while loop produces a lexicographically larger sequence. The very last iteration of std::next_permutation will return false and will convert it to a ascending sorted list. The complexity of each iteration of the while loop is **O(N)**. The total number of permutations possible is **N!**. Thus the overall complexity is **O(N*N!)**.

3 Trickier STL

Consider a GUI for selecting multiple elements in an ordered collection (e.g. a list of files in Finder or File Explorer). A common task that a user would like to perform with such a GUI is to select many elements, then drag and drop them elsewhere. The destination might be the same collection. Let us model the ordered collection as an iterator range [f, l) and the dropping location as an iterator p. We require that p is reachable from f and l is reachable from p. Let us also assume that we have a predicate s that will tell whether an element is selected or not. For example, s(*f) is true if the first element is selected, and false if not.
Implement a generic function that performs the above "drag and drop" manipulation for the range [f, l), given p and s. Its prototype should be:

```
template <typename BidirectionalIterator, typename UnaryPredicate>
void dragdrop(BidirectionalIterator f, BidirectionalIterator l,
             BidirectionalIterator p, UnaryPredicate s);
```

So after a call dragdrop(f, l, p, s) it must be that

- all elements for which s holds are consequtive in [f, l);
- the elements that came before p in the original sequence and for which s does not hold are a prefix of [f, l);
- the elements that came after p in the original sequence and for which s does not hold are a suffix of [f, l);
- the relative order of the elements that satisfy s is the same in the original sequence as it is in the resulting sequence; and
- the relative order of the elements that do not satisfy s is the same in the original sequence as it is in the resulting sequence.

This test program shows how the dragdrop algorithm could be used. Per the predicate, all elements greater than or equal to 10 are considered to be selected. The drop location is between values 60 and 70. Determine whether the template can be instantiated with:

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <initializer_list>

// dragdrop implementation here

int main() {
  std::vector<int> v = { 1, 20, 3, 40, 5, 60, 70, 80, 9 };

  dragdrop(v.begin(), v.end(), v.begin()+6, [](int i) { return i >= 10; });

  std::for_each(v.begin(), v.end(), [](int i) { std::cout << i << "␣"; });
}
```

The output of the program is as follows:

```
1 3 5 20 40 60 70 80 9
```

Ans - We can use stable::partition twice. The first half calls the inverted unary predicate, the second half calls the normal unary predicate. To invert the unary predicate, we need to know its argument_type. This can be indirectly gotten through the bidirectional iterator's value type. std::not1 takes a std::function to get its inverse. To cast the unary predicate to a std::function, we use the return type as bool and its value type as the one be got from the iterator.
We can also implement the needed operations by using a series of std::rotate calls.

4 Iterator traits

Write a function print_category that can be used to print the category of an iterator. The function should have the following prototype:

```
template <class Iterator>
void print_category(Iterator x);
```

If x is an InputIterator, it should print Input Iterator, for ForwardIterator, it should print Forward Iterator, and so on.

- What is printed in response to the following?

  ```
  char a[10];
  print_category(a + 10);
  ```

- What is printed in response to the following?

```
std::list<int> a;
print_category(a.begin());
```

- Create a test program that will cause each kind of iterator to be printed.

Ans - We can use iterator traits to figure out the iterator category. ‘std::iterator_traits’ is the type trait class that provides a uniform interface to iterator properties. The class defines several types that correspond to typedefs provided by ‘std::iterator’.

```
template <class Iterator>
void print_category(Iterator x) {
   print_category(x,
   typename std::iterator_traits<Iterator>::iterator_category());
}
```

‘std::iterator_traits<Iterator>::iterator_category’ is an alias to one of the five available tag type -

- input_iterator_tag corresponds to InputIterator
- output_iterator_tag corresponds to OutputIterator
- forward_iterator_tag corresponds to ForwardIterator
- bidirectional_iterator_tag corresponds to BidirectionslIterator
- random_access_iterator_tag corresponds to RandomAccessIterator.

These iterator category tags carry information that can be used to pick the most efficient approach. For example, if a particular container gives us a RandomAccessIterator, we can then use std::sort versus when we only get a ForwardIterator (as in the case of a list).