SOME EFFECTIVE OOP EXAMPLES

L. S. Tang
Department of Mathematics and Computer Science
Western New England College
Springfield, MA 01119

1. INTRODUCTION

Object-oriented programming has become increasingly important in undergraduate computer science education. However, good examples are difficult to find. Many textbook examples of inheritance and polymorphism are usually unrealistic and can not convince students that object-oriented programming is useful in real world applications. Good multiple inheritance examples are even harder to find.

This paper will present two examples that we feel can convince students of the benefits of object-oriented design and spur their interest. Both examples are well-known and fun to program. We first describe a sorting animation. In this project we will ask students to implement animation of various sorting algorithms, and for each sorting algorithm the user can choose one of two methods of animation. We will show that polymorphism and multiple inheritances would be most natural in the implementation design.

The second example shows an object-oriented approach to solve the classical Knight’s tour problem. A traditional solution uses the backtracking method. In this method the number of possible states increases exponentially with respect to the depth of search. A heuristic search can efficiently select a more promising move. In the object-oriented approach we will combine the traditional backtracking method and the heuristic search using inheritance and polymorphism.

This paper is intended to serve as case studies in object-oriented design and programming. We choose C++ as the implementation language. The implementation is readily transposed to other object-oriented languages.

2. SORTING ANIMATION

Computer algorithms are often difficult to explain by merely describing the procedure in textual form. Tracing through the code can be very boring to students. A method that can effectively spur students' interest is graphics animation. Graphics visualization can clarify how and why an algorithm works. It has been used effectively in teaching computer science courses [1]. A classical method for animating sorting algorithms uses vertical bars to represent array elements, for which each value is reflected by the height of the corresponding bar. When the algorithm interchanges two elements, we redisplay the bars corresponding to the two array elements being swapped. This method helps us to illustrate the dynamic movement of array elements. This is often called the **Bars View** of the sorting animation. To add some clarity to the animation and to provide more information, we can use colors to highlight essential areas of the array elements. As an example, consider the Insertion sort. We can display the unsorted porting of the array by some color (shown as yellow in Figure

1 below), the sorted portion by another color (gray in Figure 1) and the new element to be inserted is highlighted by a brighter color (red in Figure 1). This use of different colors can draw our attention to the dynamic movement that shows how each new element is inserted.

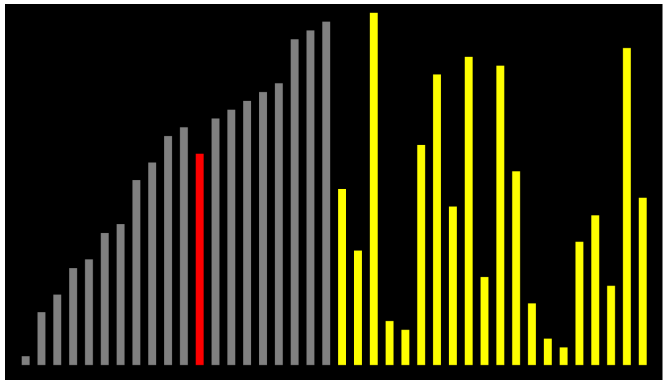


Figure 1. Insertion sort - Bars View.

Another traditional method is the **Dots View**, for which each array element is represented by a dot (a small circle or a square) on the screen. We can think of these dots as the tops of the bars in the Bars View. So, the vertical position of the dot on column i represents the relative value of a[i]. When the array is sorted, these dots will cluster to the diagonal of the screen to represent the increasing order of array elements. Figure 2 shows a screen of the running Quicksort in Dots View. The Bars View and the Dots View each has its advantages in illustrating certain aspect of the sorting algorithm. For the benefits of each view and the use of colors please see [5].

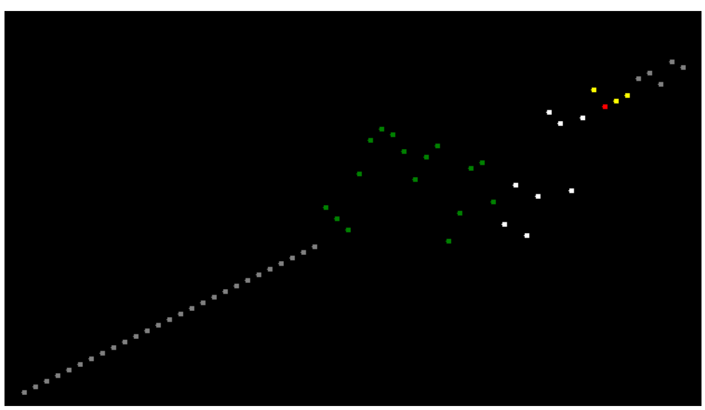
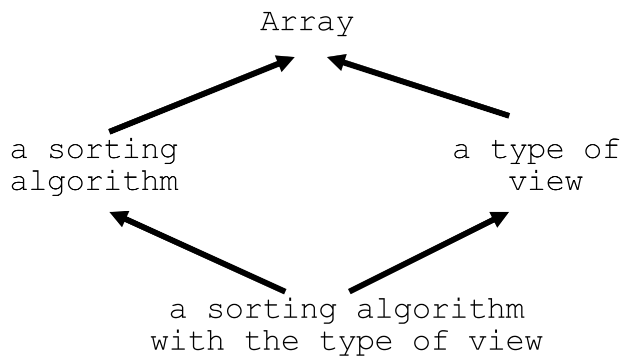


Figure 2. Quicksort - Dots View.

Instead of just showing students the animation, we ask students to program it, so they get even better understanding of each algorithm and the animation process.

IMPLEMENTATION. In this project we want to implement the animation for several sorting algorithms on an array of integers, and each in two possible views. Also, we would like to have the capability to allow the user to choose an algorithm in either view at the **run time**. Because the dynamic nature of the animation program, it is best implemented in an object-oriented language. By examining the problem statement using a technique such as **Noun extraction**, we can easily identify the array as an object. The method to sort the array object depends on the algorithm we choose. So we declare it as an **abstract virtual** method. To implement a particular sorting algorithm, we simply derive a subclass from the array class and define the sort method. The type of view defines how we display array elements, and it is independent to the sorting algorithm. We will define two abstract virtual methods, display and displayAll. The display method displays an array element using a specified color, and the displayAll method displays all array elements using a specified color. The implementation of these two methods is provided by a subclass. Once all sorting algorithms and the two views are defined, we can combine a sorting algorithm with a particular view by a **multiple inheritance.**




```
class Array
{
protected:
     int  *a;  // dynamic array, will be created by constructor
     int  size;
     virtual void displayAll(int color)      {}  // abstract
     virtual void display(int i, int color)  {}  // abstract
     void         hide(int i);
     void         swapAndDisplay(int i, int colori,
                                 int j, int colorj);
     // ... other members
public:
     Array(int n, int *x);     // constructor
     ~Array();                 // destructor
     virtual void sort() {}    // abstract
};
```

The Array class is abstract and all virtual methods are abstract to force the implementation in subclasses. The hide method simply calls display method to display the ith element using the background color. The swapAndDisplay method will be called by a sorting algorithm to swap two elements and redisplay them in some given colors.

```
void Array::swapAndDisplay(int i, int colori,
                           int j, int colorj)
{
     int  temp;

     hide(i);
     hide(j);
     temp = a[i];
     a[i] = a[j];
     a[j] = temp;
     display(i, colori);
     display(j, colorj);
}
```

To implement a particular sorting algorithm, we can simply derive a class from the Array class and redefine the sort method. As an example, here is how we can implement the Insertion sort animation:

```
class InsertionSort : virtual public Array
{
public:
     InsertionSort(int n, int* x);  // constructor
     void sort();      // define the sort in this class
};

...

void InsertionSort::sort()
{
     int   i, j, newitem;

     displayAll(YELLOW);
     display(0, GRAY);
     for (i = 1; i < size; ++i)
     {
          display(i, RED);
          newitem = a[i];
          j = i-1;
          while (j >= 0 && newitem < a[j])
          {
               swapAndDisplay(j, RED, j+1, GRAY);
               --j;
          }
          display(j+1, GRAY);
     }
}
```

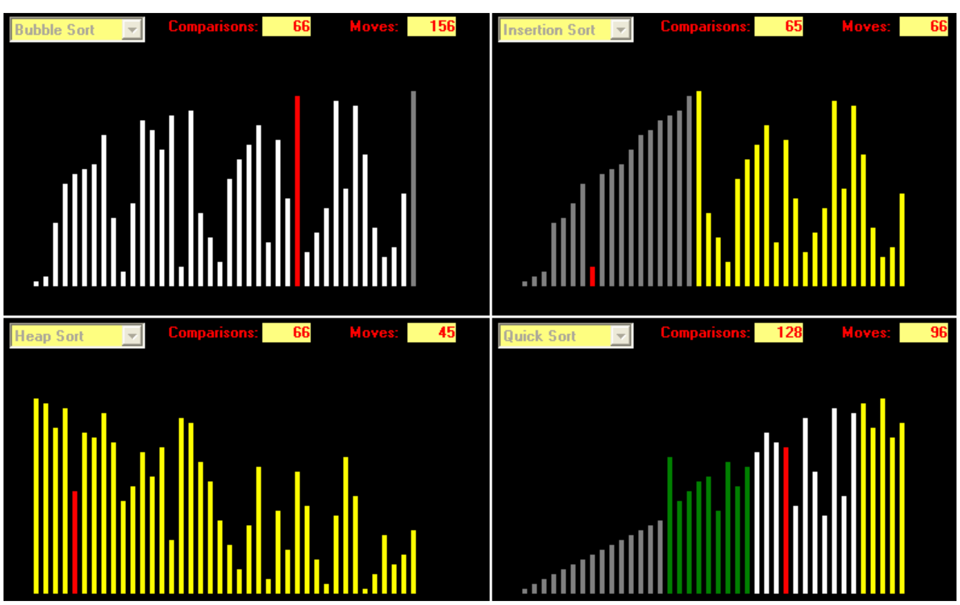
Note that the sort method calls displayAll and display methods which are not defined in this class or the base class. They will be handled by another subclass of Array. For example, to implement the Bars View, we can derive a class from the Array class and define the displayAll and display methods:

```
class BarsView : virtual public Array
{
private:
     // ... variables to record screen information
protected:
     void displayAll(int color);
     void display(int i, int color);
public:
     BarsView(int n, int *x);
};
```

We have omitted implementation details for brevity. Now, suppose we want to implement the Insertion sort in Bars View, we can derive a class from the InsertionSort and BarsView classes by multiple inheritance:

```
class InsertSortInBarsView : public InsertionSort,
                             public BarsView
{
public:
     InsertSortInBarsView (int n, int *x);  // constructor
};
```

The constructor of InsertSortInBarsView simply calls the constructor of each base class. No additional programming is required. In this object-oriented approach, we can easily add new sorting algorithms or different type of views. The benefit of inheritance and polymorphism would be clear to the students. Since the project also involves graphics animation, it can stimulate students' interest to learn and to program. For a language that supports multi-threads, we can enhance the project to animate multiple sorting algorithms on one screen so they can be compared visually. Object-oriented programming makes such enhancement easy to implement.



3. KNIGHT'S TOUR

In this classical problem, we have an nxn chessboard with n^2 positions. A knight is placed on one position. We are to move the knight, according to the rules of chess, such that each position of the chessboard is visited exactly once.

Backtracking is a traditional method for solving this problem. It is well-known that backtracking method suffers efficiency problem for which the search tree may grow exponentially with respect to the depth of search. A heuristic search can efficiently select a more promising move. In this project we would like to implement both the traditional backtracking method and a heuristic search and allow the user to choose and compare their performance at runtime.

Reiss [6] gives a detailed analysis of this problem. Our solution, however, follows more closely to Wirth's solution [8], but it uses an objected-oriented approach and makes use of standard library of modern objected-oriented language.

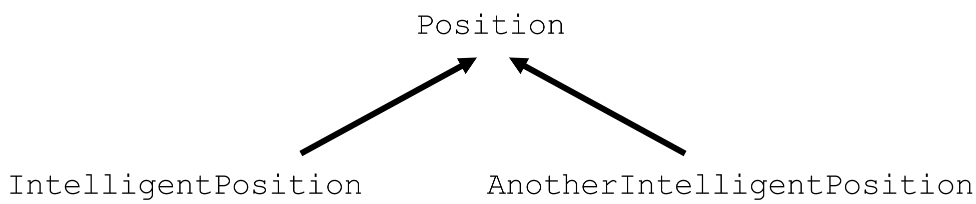
By analyzing the problem statement, we can identify two classes: **Position** and **ChessBoard**.

The **Position** class defines a square position on a Knights tour chessboard, and all possible moves from this position. For a heuristic solution we need to order moves so more promising moves can be tried first. We choose to use C++ STL container class **multimap** to hold these moves. Since a multimap object's insert method uses a key value to order its elements, we use a method valueOf to calculate and return the key value of a move. To allow more intelligent heuristic search, we declare this method as **virtual** so a derived class can override it and multimap's insert method would call the proper version of valueOf to order its elements.

```
struct OneMove
{
     int  newRow;      // position of new (row, column) of a
     int  newColumn;   // possible move from the current position
};

typedef multimap<int, OneMove> setOfMoves;
...
class Position
{
private:
     int         rowPos, columnPos;
     setOfMoves  moves;  // set of all moves from this position
     int         numMoves;
     virtual int valueOf(OneMove);
public:
     Position(int row, int column);
     bool    hasMoreMove();
     OneMove nextMove(); // What is the next possible move?
     ... other data members and methods
};
```

The constructor will set up all possible moves from this board position, by calling the moves.insert method. The insert method would add a move into the sorted list by the order of key value returned by the valueOf method. For the normal backtracking method, all moves are of the same key value, and for an intelligent method, a derived class may override it with a heuristic value. To implement an intelligent heuristic search we simple derive a class from Position and redefine the valueOf method.



The ChessBoard class uses an array to keep track all board positions.

```
class ChessBoard
{
private:

     int       size;        // Size of the chessboard.
     Position  *positions;  // array of all positions on the board

public:

     ChessBoard(int n = 5, int method = 1);
     void  markAMove(OneMove);
     void  eraseAMove(OneMove);
     Position  positionAt(OneMove);
     bool  tourDone();

     ... other data members and methods
};
```

The main program would be similar to the recursive solution given by Wirth to find a tour from a given move position on the board:

```
bool tryNextMove(ChessBoard &board, OneMove m)
{
     Position   currentPos = board.positionAt(m);
     OneMove    nextMove;
     bool       successful = false;
     while (!successful && currentPos.hasMoreMove())
     {
          nextMove = currentPos.nextMove();
          board.markAMove(nextMove);
          if (board.tourDone())
               return true;
          else
```

```
            {
                 successful = tryNextMove(board, nextMove);
                 if (!successful)
                      board.eraseAMove(nextMove);
            }
       }
       return successful;
  }
```

By comparing this object-oriented approach and the traditional solution, we can conclude that the object-oriented approach is cleaner and much easier to understand from the students' point of view. Various parts of the program are decomposed into different components. Also, OOP makes it easier to extend the functionality of the program. To add more fun to the project, we can implement graphics animation to the program.

4. SUMMARY

To teach object-oriented concept without proper examples can be counter-productive. However, just like teaching and learning of any subject, good examples of OO are usually difficult to find. This paper has presented two examples that we had used successfully in teaching students the benefits of object-oriented concept and spur their interest in learning.

5. REFERENCES

1. Marc Brown, Algorithm Animation, *MIT Press*, 1988.

2. Marc Brown and John Hershberger, Color and Sound in Algorithm Animation, *SRC Reports*, No. 76a, DEC Systems Research center, 1991.

3. Marc Brown and Robert Sedgewick, Techniques for Algorithm Animation, *IEEE Software*, 2(1), pp. 28-39, 1985.

4. Bjarne Stroustrup, *The C++ Programming Language*, second edition, Addison-Wesley, 1991.

5. L. S. Tang, *Colorful Sorts*, The Journal of Computing in Small Colleges, 9(2), pp. 28–34, 1993.

6. Steven Reiss, *Software Design with C++*, Wiley, 1999.

7. Martin Gardner, *Mathematical Games: Problems that are built on the knight's move in chess*, Scientific American, pp. 128-132, Oct. 1967.

8. Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.

9. Mel Ó Cinnéide and Richard Tynan, *A Problem-Based Approach to Teaching Design Patterns*, SIGCSE Bulletin, 36(4), pp. 80-82, 2004.