UNBC Laboratory Assignment
Computer Science 200—Fall 2007

Dijkstra's Dutch Flag Problem

Goals:

The goals of this laboratory exercise are three-fold:

- To practise using loop-invariants.
- To practise timing code to check its asymptotic *Theta*-behaviour.
- To gain confidence in interfacing with libraries supplied by someone else.

Due Date:

This assignment is due *Tuesday, 23 October* at the beginning of class.

The Dutch Flag problem

The Dutch flag problem is due to Dijkstra, and illustrates the value of carefully thinking through loop invariants.

The problem goes as follows. You are given an array (actually a `vector`) filled with n red, white, and blue objects; and you are to sort it so that the red objects are at the top, and the blue ones at the bottom. Now, in general sorting requires $\Omega(n \log n)$ average time, but in this particular case, because there are only three possible kinds of objects, you can achieve $\Theta(n)$ worst-case time.

Your assignment is to think hard about this problem, preferably by thinking about loop invariants; code your solution; document in your code all loop invariants that are important; and produce graphs of the time behaviour of your algorithm to verify that your algorithm is indeed $O(n)$.

The `RedWhiteBlue` class

To ensure that you think about red, white, and blue objects abstractly, you are supplied with a `RedWhiteBlue` class (see Figure 1). This class is given to you as `.o`- and `.h`- files, so you must write your program to run on `galaxy.unbc.ca` and link your code there.

The `RedWhiteBlue` class supplied is deliberately quite minimalistic. In fact it does not contain a zero argument constructor. If you really want, you can copy construct individual values from the constants `RedWhiteBlue::red`, `RedWhiteBlue::white`, and `RedWhiteBlue::blue`. However, you are more likely to want to create a random vector of `RedWhiteBlue` objects, and the static member function `RedWhiteBlue::makeRandomRWBVector` does this. It takes as arguments the number of objects of each colour that you want.

```
#include <vector>
class RedWhiteBlue_Implementation ;

class RedWhiteBlue
{
public:
    // ‘‘global’’ constants
    static RedWhiteBlue red ;
    static RedWhiteBlue white ;
    static RedWhiteBlue blue ;

    // construction and assignment
    static vector<RedWhiteBlue> makeRandomRWBVector(int r, int w, int b) ;
    RedWhiteBlue(const RedWhiteBlue&) ;
    RedWhiteBlue& operator= (const RedWhiteBlue&) ;

    void swap(RedWhiteBlue&) ;
    ~RedWhiteBlue() ;


    // properties
    bool isRed() const ;
    bool isWhite() const ;
    bool isBlue() const ;
private:
    RedWhiteBlue(const RedWhiteBlue_Implementation*) ;
    RedWhiteBlue_Implementation * p_impl ;
} ;
```

Figure 1: RedWhiteBlue class declaration

Library locations

The `RedWhiteBlue` class declaration can be found in `RedWhiteBlue.h` on `galaxy.unbc.ca` in the directory `/export/student_home/home/casper/C++/200/RedWhiteBlue`. A `tar`'d and `gzip`'d file can be found on David Casperson's website. A compatible object file `RedWhiteBlue.o` can be found in the same directory. The approximate text of `RedWhiteBlue` is shown in Figure 1.

Check List:

Make sure that you have:

⇒ written a C++ function with signature

```
void sort(vector<RedWhiteBlue>& data) ;
```

that takes a vector of unsorted data and returns a vector of sorted data.

⇒ verified with your stop watch that it runs in $O(n)$-time, and plotted your results. It should only use constant extra storage (or at least $o(n)$ extra storage!), but you do not need to verify this experimentally.

⇒ shown in your output how you have linked your file against the class supplied.

⇒ documented in your code the invariants for any and all loops inside your sorting function.

⇒ handed in nicely graphed data produced by your stop watch measurements.