Miscellaneous Études

Due Date:

This assignment is due 28 March 2007 *at the beginning of lecture.*

Static member functions, constructors, and assigment operators

One reason why you might want to overload the assignment operator and copy constructor is when you have a class where each object has a unique identification number.

For this exercise imagine that you wish to model `SpeedingTickets`. Each speeding ticket should have a unique id number (and, for this exercise, this means unique even if created by a copy constructor). It should be possible to create an array of `SpeedingTicket`s and have each `SpeedingTicket` get a unique id. Each ticket should have a fine. There should a be a static member function that tells you how many ticket objects currently exist.

Programming exercises:

Create a `SpeedingTicket` class that has:

1. appropriate constructors and destructors,
2. a static member function that sets the default fine for newly created tickets,
3. an overloaded assignment operator that does not change the ticket id of the ticket being assigned to,
4. a copy constructor that guarantees the uniqueness of ticket ids,
5. a static member function `int numberOfTickets` that returns the number of currently existing `SpeedingTicket` objects,
6. an `int getID() const` member function that returns the unique id number of a ticket,
7. `getFine` and `setFine` member functions,
8. an overloaded `<<` operator,[1] and
9. whatever other member or non-member helper functions or variables you need.

Create test program(s) that show:

1. that it is possible to create an a array of tickets, and that each ticket created has a different id;
2. that the static member function that sets the default fine works correctly;
3. that the copy constructor works;
4. that the count of currently existing tickets works; and
5. that the `getFine`, `setFine`, and overloaded `<<` operator work.

+This **.pdf** created March 19, 2007.
[1]overloaded by a non-member function.

SmartDatePointer.h
```
#if !defined(SmartDatePointer_INCLUDED)// This file is really -*-C++-*-
//      Description:  header comments here

class Date ;

class SmartDatePointer
{
public:
    SmartDatePointer() ; // creates a bad pointer
    SmartDatePointer(Date array[], int size) ; // points at elt 0
    SmartDatePointer(Date array[], int size, Date * ptr) ;
    // points at a given element of an array
    void setArrayAndSize(Date array[], int size) ; // change the array

    // properties
    Date* base_array_ptr () const ;     // return a pointer to element 0
    int base_array_size () const ;      // return the size of the array
    bool isValid() const ;              // return true if current pointer is ok
    Date& operator*() const ;           // dereference if valid
    Date* operator->() const ;          // yield a pointer if valid

private:
    Date * array_ptr ;
    int array_size ;
    Date * ptr ;
} ;// end class SmartDatePointer

SmartDatePointer operator+(SmartDatePointer const&, int) ;
SmartDatePointer operator+(int, SmartDatePointer const&) ;
SmartDatePointer operator-(SmartDatePointer const&, int) ;
int operator-(SmartDatePointer const&, SmartDatePointer const&) ;

#define SmartDatePointer_INCLUDED
#endif// !defined(SmartDatePointer_INCLUDED)
```

Figure 1: Interface for a SmartDatePointer class

Overloading the arrow operator

The arrow operator (`->`) is a little strange. It must be overloaded by member function and its return type must be a pointer. Although it is technically a binary operator, it must be overloaded as if it were unary. Suppose for instance that we have a `SmartDatePointer` class with a member function

```
Date * operator->() const
```

and we have an object

```
SmartDatePointer sdp ; // presumably initialized later!
```

If we write something like "`sdp->setDay(12)`", the compiler converts this to `(sdp.operator->()) -> setDay(12)`. The second arrow here is an HTG[2] arrow operator.

[2]Honest To Gertrude

The main reason for overloading the “`->`” operator is when you wish to create a class of objects that behave like pointers, but have extra functionality. In the following exercise, you write a `SmartDatePointer` class whose objects behave almost like `Date *` objects, but which are only allowed to point at elements of an array of `Date` objects, and which check to make sure that bad pointer arithmetic is never done.

Programming exercises:

Write a `SmartDatePointer` class that has an interface similar to that shown in Figure 1. Write a driver program that tests your `SmartDatePointer` class on a small array of `Date` objects, including a test of the `->` operator. Your `SmartDatePointer` class doesn't have to have exactly the same class declaration, but it must have similar functionality.

Decide on a consistent strategy to adopt when an illegal pointer operation is attempted. This can be as simple as printing an error message and exiting the program.

For bonus marks, add all of the other operators that pointers normally have (for instance, `++` prefix and postfix, `[]`, and so on).