Programming Project 6

Due date: Tue 4/27 11:00 pm

Introduction

In this project you will write a facility that performs automatic garbage collection using reference counts albeit in a limited context. Since we don’t have as much flexibility as the compiler we will restrict the application to one template type with no internal pointers. An additional small project (for extra credit) extends this to handle list nodes. Having done this one can see how the general problem, handing arbitrary type descriptors, can be solved.

Implementation Details

Our templates will assume a class T with no member functions. To simplify things you can think of T being instantiated to basic types e.g. int or char. The facility will be packaged in two classes with the following interface:

```cpp
template <class T>
class RefCountT {
    private:
        ...
    public:
        RefCountT();
        ~RefCountT();
        RefCountT(RefCountT & r);
        void operator=(RefCountT & r);
        void inc();
        void dec();
        bool zeroCount();
        void setVal(T val);
        T getVal();
};

template <class T>
class RefCountPtrT {
    private:
        RefCountT<T> *ptr;
        ...
    public:
        RefCountPtrT();
        ~RefCountPtrT();
        RefCountPtrT(RefCountPtrT & r);
        void operator=(RefCountPtrT & r);
        void makeNull();
        bool isNull();
        void allocNew();
```
void setVal(T val);
T getVal();
};

The class `RefCountT` holds the object of type `T` and the reference count. The functions are as follows:
- `inc()` increments the count,
- `dec()` decrements it,
- `zeroCount()` returns true if the count is zero (and thus indicates that the `RefCountT` object can be deleted),
- `setVal()` assigns the value `val` to the `T` object, and
- `getVal()` returns the value of the object.
In addition you will need to write a constructor, destructor, copy constructor, and overload the assignment operator. Recall that the copy constructor is used when a new object is created and initialized from an existing object. The assignment operator is used when an object is copied into an existing object. You should handle both data and counts correctly for these operations.

The class `RefCountPtrT` represents a pointer to such objects. Its data member is a pointer to `RefCountT`. The access and use of these pointers is limited to the following interface:
- `makeNull()` assigns NULL to the pointer,
- `isNull()` returns true if the pointer is NULL,
- `allocNew()` replaces the standard `new` function; it allocates space for the `RefCountT` and makes the pointer point to it.
- The functions `setVal(T val)` and `getVal()` assign and return a value to the `T` object.
Here as well you will need to write a constructor, destructor, copy constructor, and overload the assignment operator.

Notice that all the operations above must implicitly support correct management of the objects and counts. For example, suppose that we have two `RefCountPtrT` objects `p1` and `p2` (which are our pointers), and we assign `p1=p2`. Then we must decrement the count for `p1`'s object, delete the object if the count is now 0, copy `p2` to `p1`, and increment the count for `p2`'s object. In this way we get automatic memory reclamation on the fly and the programmer never uses the delete operation. Other member functions must similarly manage the counts and possibly deallocations.

For both classes you may add any other private member functions that will help in simplifying the implementation. The classes should crash gracefully if the program tries an illegal operation like trying to assign or read from a NULL pointer; in this case the program should print an error message and exit.

The required behavior is illustrated by the following code. You can follow object counts and verify that allocations and deallocations happen in the locations as indicated.

```cpp
RefCountPtr<int> p1, p2, p3;
p1.allocNew();  // allocate object #1
p1.setVal(5);
p1.makeNull();  // deallocate object #1

p1.allocNew();  // allocate object #2
p1.setVal(5);
cout << "p1 Value is " << p1.getVal() << endl;  //prints 5

p2.allocNew();  // allocate object #3
p2 = p1;  // deallocate object #3
p2.setVal(1+p2.getVal());
cout << "p2 Value is " << p2.getVal() << endl;  //prints 6

p3.allocNew();  // allocate object #4
p1 = p3;
p2.makeNull();  // deallocate object #2
```

Your Task

1. Implement the 2 classes in files: `RefCountT.h`, `RefCountT.cpp`, `RefCountPtrT.h`, `RefCountPtrT.cpp` with the usual arrangement.

Note that we cannot run separate compilation with templates using g++. Therefore, your main program needs to include the cpp files rather than the header files.
2. Write a simple main program using these classes so that all interface functions are exercised (you don't have to worry about (de)constructors only the other functions). Document the program to indicate which functions are used at each step. Your annotation should include counts and allocation as well as mention specific functions that are used and/or different cases in these functions. Put this program in a file pp6.cpp

Files

The headers and the main program illustrated above are provided in /comp/80/files/pp6/

Submitting your program

Make sure that your code is easily readable and well documented. The grade will be based both on correctness and on readability. You must use filenames as above. Your files should have include statements so that everything compiles using just g++ pp6.cpp. Assuming your code is in the current directory on one of our Sun workstations or servers, you should submit by typing

provide comp80 pp6 RefCountT.h RefCountT.cpp RefCountPtrT.h RefCountPtrT.cpp pp6.cpp

If you finish only part of the project you may submit any subset of these files.

Of course your classes should work well with any main program using the interface correctly and we will test your code on using such programs.

For Extra Credit

This part is optional and will award you up to 10 extra points. Extend the classes above as follows: RefCountT will now include in addition a RefCountPtrT data member. Thus RefCountT can be used as a node in a linked list. The interface for RefCountPtrT should be extended with functions giving access to the “next” pointer as follows:

RefCountPtrT getnext();
void setnext(RefCountPtrT & n);

where getnext returns the value of the pointer “next” and setnext assigns the value of n to it. For example the following code (provided in the files directory)

RefCountPtrT<int> p1,p2,p3;
p1.allocNew();
p1.setVal(5);

p2.allocNew();
p2.setVal(6);
p1.setnext(p2);

p2.allocNew();
p2.setVal(7);
(p1.getnext()).setnext(p2);

p3 = p1;
while(!p3.isNull()) {
    cout << "List Value " << p3.getVal() << endl;
p3 = p3.getnext();
}

will print
Please use the same file names as above for the classes. You should also write a main program in a file pp6b.cpp and annotate it appropriately. To submit use:
provide comp80 pp6b RefCountT.h RefCountT.cpp RefCountPtrT.h RefCountPtrT.cpp pp6b.cpp

For Extra Credit

This part is optional and will award you up to 10 extra points. It can only be done in combination with the previous part. Write a linked list class based on the previous part. You class should maintain an integer list in sorted order without duplicates and support the operations: insert(int), delete(int) and member(int) with the obvious meaning, and howmany() returning the size of the list. You should also write a simple main program which illustrates how the class works.
We are assuming that this new class will work well with the files from the previous part. Put the new parts in MyList.h, MyList.cpp, pp6c.cpp and submit using
provide comp80 pp6c MyList.h MyList.cpp pp6c.cpp