Assignment 4: Tombstones

COMP 3010 – Organization of Programming Languages
April 2019

[**Note:** *This assignment is adapted from the original version authored by Prof. Michael Scott.*]

Problem description

Unlike most modern object-oriented languages, C++ does not provide automatic garbage collection; in fact, completely accurate collection is not possible given the lack of complete type safety inherited from C.

As mentioned in the PLP textbook, it is possible in C or C++ to implement *conservative collection* [1], which reclaims any object whose address does not appear as a bit pattern in any global variable, stack variable, or potentially reachable heap variable.

[1] http://www.hpl.hp.com/personal/Hans_Boehm/gc/

In this assignment we will explore a different approach, which leverages templates and operator overloading in C++ to implement *tombstones* (described on pages C-144–C-146 of the Supplementary Sections downloadable from the PLP companion site [2], and again briefly on page 389 and 391 of the main text). The programmer remains responsible for manual storage reclamation, but the library package catches all dangling references and all memory leaks of noncircular structures. (There's no good way to catch leaks of cycles, given that C++ is not type safe.) Such a library can be valuable during development and testing. If performance is critical, tombstones can then be turned off during production use. (This approach to checking, famously likened [3] to the sailing enthusiast who takes off his or her life jacket when heading out to sea, isn't necessarily a good idea, but it's certainly common practice.)

[2] http://booksite.elsevier.com/9780124104099/

[3] http://flint.cs.yale.edu/cs428/doc/HintsPL.pdf

The details

Your task is to build a template class that behaves much like a pointer, but uses tombstones for error checking:

```
template <class T>
class Pointer {
public:
    Pointer<T>();                                   // default constructor
    Pointer<T>(Pointer<T>&);                        // copy constructor
    Pointer<T>(T*);                                 // bootstrapping constructor
        // argument should always be a call to new
    ~Pointer<T>();                                  // destructor
    T& operator*() const;                           // dereferencing
    T* operator->() const;                          // field dereferencing
    Pointer<T>& operator=(const Pointer<T>&);       // assignment
    friend void free(Pointer<T>&);                  // delete pointed-at object
        // This is essentially the inverse of the new inside the call to
        // the bootstrapping constructor.  It should delete the pointed-to
        // object (which should in turn call its destructor).
    // equality comparisons:
    bool operator==(const Pointer<T>&) const;
    bool operator!=(const Pointer<T>&) const;
    bool operator==(const int) const;
        // true iff Pointer is null and int is zero
    bool operator!=(const int) const;
        // false iff Pointer is null and int is zero
};
```

You'll probably also want the following, to allow `int`-to-`Pointer` comparisons (not just `Pointer`-to-`int`).

```
template <class T>
bool operator==(const int n, const Pointer<T>& t) { return t == n; }

template <class T>
bool operator!=(const int n, const Pointer<T>& t) { return t != n; }
```

Please name your source files `tombstones.h` and (if you need it) `tombstones.cc`. Programmers should be able to use your Pointer class via:

```
#include "tombstones.h"
```

Ideally, you should like to be able to take an arbitrary (correct) pointer-based program, replace instances of `T*` with `Pointer<T>`, and have it compile and run correctly (it may run a bit slower, and use a bit more memory). If the program uses a dangling reference, however, you should get a useful run-time error message instead of bizarre behavior of a segmentation fault or unexpected behavior.

Division of labor and submission procedure

You may work alone on this project, or in teams of two. Be sure your write-up (README file) describes any features of your code that the TA might not immediately notice.

A sample driver program `A4-tests.cpp` is provided on Blackboard. Your implementation will be complete when:

- none of the errors 1 through 9 are produced;
- all of the values in the output are correct per the output messages; and
- the driver program produces an error message when dereferencing the dangling pointer at the end (and exits before outputting Error 10!).

Also include copies of whatever test programs you have written yourself and used to exercise your code, as well as a `Makefile` that will compile your own test programs when the command "`make test`" is executed.

To turn in your code, use the following procedure, which will be the same for all assignments this semester:

1. Your code should be in a directory called "`<YourName>_OPL_A4`" (for example, "`TomWilkes_OPL_A4`"). Put your write-up in a `README.txt` or `README.pdf` file in the same directory as your code.
2. In the parent directory of your A4 directory, create a `.tar.gz` file that contains your A3 directory (e.g., "`TomWilkes_OPL_A4.tar.gz`"). If you don't know how to create a `.tar.gz` file, read the `man` pages for the `tar` and `gzip` commands (or ask a friend!).
3. In the page for Assignment 4 on Blackboard, use the Submit button to upload your `.tar.gz` file. When you submit, give the name of your partner (if you had one) in the submission comments field.