Controlling the Destruction Order of Singleton Objects

Evgeniy Gabrilovich
gabr@acm.org

Abstract

The Singleton pattern [1] is a solution to (some of) the drawbacks of using global variables. Among its advantages is that the instance is always created prior to being referenced (this effectively solves the problem of initialization order when several interdependent instances are involved). This article examines some of the existing Singleton realizations in C++ and their drawbacks, and presents a solution to the complement problem, namely, the destruction order of global instances. To this end, I propose to use a Destruction Manager, so that all the singleton objects register with it, specifying interdependencies between them. Following [2], the Destruction Manager is analyzed as a composite design pattern.

An improved Singleton implementation

The Singleton object creational pattern [1] is a superior implementation of the notion of global objects. Its major benefits are

1) limitation on the number of class instances that can be created;
2) instances are only created if actually needed;
3) the instance is always created prior to being referenced (this effectively solves the problem of initialization order when several interdependent instances are involved).

The sample code provided in [1] ignores the problem of global objects destruction, relying on the operating system to release the memory they occupy. The memory leak itself may not be so damaging, as the OS indeed reclaims the dynamically allocated memory when the process terminates. But if the destructors are responsible for releasing system-wide resources, things can get pretty messy if the resource allocation and deallocation do not follow a disciplined pattern. In [3], Scott Meyers suggests an alternative Singleton implementation to the one suggested by the GOF in [1], where the object instance is defined static in a dedicated function which returns a reference to it. Such definition invokes the singleton destructor prior to program termination, thus preventing memory and resource leak, but tightly binding the singleton object with the enclosing function.

Irrespective of the number of calls to the constructing method of the Singleton, only one instance of the class is created. Since all the clients of the class share this lone instance, the design should impose a controlled protocol regarding the deletion of this object. Although a number of efforts have been made to formalize this approach [4], none of them addresses the issue of the destruction order in the presence of several singleton objects depending on each other.

While encapsulation of globally available objects in functions indeed solves the problem of mutual initialization order, what happens if the destruction order matters too? Suppose there is a global Logger object, and destructors of other global entities must record with it the resources they release. Obviously, the Logger should be the last one to be destroyed. If the language rules destroy the Logger first, other objects

might unknowingly attempt to use it, leading to unpredictable (and most probably disastrous) consequences.

Listing 1 shows an alternative Singleton implementation, which will later serve the basis for a variation with certain additional properties. Let us consider the features of the proposed solution.

- The resource management of the Singleton is performed through the use of C++ Standard Library [5] implementation of the smart pointer -- the auto_ptr<> class template. The auto_ptr owns the object pointed to by its data member, and is responsible for its deletion. To facilitate this scheme, an auxiliary private function of class Singleton (get_instance()) defines a static auto-pointer to the actual object, which serves as a "proxy" for the latter. The object instance owned by the auto_ptr is thus detached from the access function wrapper (instance()) that ensures its singleton properties. This scheme does not sacrifice proper memory management, but allows real objects to be destroyed in the order other than the one induced by C++ language rules, thus facilitating alternative destruction policies.

- If the auto_ptr actually gets to delete the object it owns, it needs access to its destructor, hence the friendship between class Singleton and auto_ptr<Singleton>.

- *Implementing the "instance" as a static variable of the method.*
The single instance of the auto_ptr is implemented as a static variable in the method Singleton::get_instance(). The auto_ptr in turn controls a single instance of the singleton object, which is created on the heap (using operator **new**). This provides a clean solution to the problem of initialization order, which would have cropped up if the auto_ptr were created as a static data member of the class. In [1], the GOF implement the latter approach (making the singleton instance pointer a static data member of the class), but that implementation uses a regular pointer, rather than an auto_ptr. A C++ built-in pointer has no constructor, and the Standard guarantees that a global pointer is initialized to zero. But in this case, if the static auto_ptr were made a data member, then the auto_ptr object would have been created at the global scope. The C++ Standard does not define the initialization order of global objects unless they are all defined within the same translation unit. Hence if another global object got created first, whose constructor needed a singleton, then the method Singleton::get_instance() would be invoked. Since there would be no guarantee whatsoever that the constructor of the auto_ptr had already been called, this may have resulted in the disastrous consequences of dereferencing a garbage value, as if it were a valid pointer.

- Function get_instance() is not defined inline, but rather in the implementation file. The new C++ standard [5] states that "a static local variable in a member function always refers to the same object, whether or not the member function is inline". Nevertheless, old compilers may still be in use implementing old language rules, under which each translation unit which includes the definition of such function contains its own static copy of this function, with its own copy of the variables. This issue is explained in detail in [3].

- Two public functions provide 'const' and 'non-const' access to the Singleton, by dereferencing the auto_ptr. For additional code safety, if the constant object will do the job, the corresponding function const_instance() should be used.

- A reference to the actual object (and not to the auto_ptr) is returned from the access functions, in order to conceal the implementation details from the clients.

Listing 1. *Improved Singleton implementation.*

singleton.h :

```
#include <memory>       // for auto_ptr

using namespace std;

class Singleton {
  typedef auto_ptr<Singleton> SingletonPtr;

  // the auto_ptr should be returned by reference;
  // otherwise the returned copy will receive the ownership
  // over the sole instance of the class
  static SingletonPtr& get_instance();

  // so that the auto_ptr may delete the singleton
  friend class auto_ptr<Singleton>;

  // Singleton objects should NOT be copy constructed or assigned.
  // Therefore, copy constructor and assignment operator made
  // private so that clients cannot invoke them and the compiler
  // doesn't create them implicitly.
  Singleton(const Singleton&);
  Singleton& operator=(const Singleton&);

protected:
  // the constructor is protected, so that no instances of this class
  // may be created except in the dedicated function 'get_instance'
  // or by the derived classes
  Singleton() { /* initialize the singleton object */ }

  // the destructor is protected so that nobody deletes the singleton
  // by mistake (or maliciously)
  ~Singleton() { /* destroy the singleton object */ }

public:
  static Singleton& instance() { return *get_instance(); }
  static const Singleton& const_instance() { return instance(); }
};
```

singleton.cxx :

```
#include "singleton.h"

Singleton::SingletonPtr& Singleton::get_instance()
{
  static SingletonPtr the_singleton(new Singleton);

  return the_singleton;
}
```

Singleton Destruction Manager – a composite design pattern for controlling the order of destruction

Let us return to the example where several global objects depend on one another, and thus cannot be destroyed in arbitrary order. If some global object is a client of another global object, the latter may not be destroyed until the former terminates. Otherwise, if the former inadvertently invokes a function of the latter after it ceased to exist, the aftermath may be rather gloomy.

My solution is based on the composite Singleton-Registration-Proxy pattern. I propose to use a dedicated Destruction Manager (itself implemented as a singleton) to control the order of global objects destruction. Whenever a singleton object is created, its constructor registers with the Destruction Manager, specifying in some way when this object should be destructed (relative to other singletons).

All the singletons in the program are assigned a "destruction phase[1]", such that the smaller the phase, the later the object should be destroyed. Each singleton constructor creates a 'destructor' object which encapsulates a pointer to the singleton ('this') and its destruction phase. At the end of main(), the programmer should invoke a dedicated function of the Destruction Manager. The latter sorts all the 'destructors' registered with it in the decreasing order of their phases, and then destroys the singletons in this order. The destruction is performed by releasing the singleton pointer from the auto-pointer which encloses it, and then deleting this pointer. If a singleton is not destroyed by the destruction manager (e.g., the destruction manager is also realized as a singleton, but does not destroy itself), the auto-pointer mechanism destroys the singleton object at program end (recall that the auto-pointer is defined static in a function).

In what follows I exemplify the gist of the proposed solution. The complete example code is available at the C/C++ Users Journal Web site http://www.cuj.com/code/ …

Listing 2 defines the destruction phase, which is comparable to its peers using operator>().

Listing 2. *Definition of the destruction phase.*

dphase.h :

```
class DestructionPhase {
```

[1] I assume that there are no circular dependencies between the objects, so it is indeed possible to assign each singleton an appropriate destruction phase.

```
  int m_phase;  // the smaller the phase, the later the object should be destroyed
public:
  explicit DestructionPhase (int phase) : m_phase(phase) {}

  bool operator> (const DestructionPhase& dp) const
  { return m_phase > dp.m_phase; }
};
```

Listing 3 shows the hierarchy of classes for destructor objects. I first define an abstract Destructor class, whose instances can be sorted according to their destruction phases. Class template TDestructor<> is derived from Destructor, and contains a pointer to the singleton object it is responsible for. The template parameter is instantiated to actual singleton classes that are assumed to define function destroy_instance(), which allows to delete the object inside the auto_ptr (that's why it is necessary to detach the object from its wrapper).

Listing 3. *Definition of class Destructor and class template TDestructor<>.*

destructor.h :

```
#include "dphase.h"
#include "dmanager.h"

class Destructor {
  DestructionPhase m_dphase;
public:
  Destructor(DestructionPhase dphase) : m_dphase(dphase)
  { DestructionManager::instance().register_destructor(this); }

  bool operator>(const Destructor& destructor) const
  { return m_dphase > destructor.m_dphase; }

  virtual void destroy() = 0;
};

template <class T> class TDestructor : public Destructor {
  T* m_object;
public:
  TDestructor(T* object, DestructionPhase dphase)
    : Destructor(dphase), m_object(object) {}

  void destroy() { m_object->destroy_instance(); }
};
```

The Destruction Manager is outlined in Listing 4 (some obvious details due to the Destruction Manager's being a singleton have been omitted for the sake of brevity). Singletons register their destructors with the Destruction Manager via function register_destructor(). The Destruction Manager is responsible for the memory occupied by Destructor objects, therefore, it has to delete them before it terminates.

Function destroy_objects() sorts the destructors in the decreasing order of their phases, and then consecutively destroys the singleton objects they manage. It is this function that should be manually invoked at the end of main() to ensure proper destruction order of the program global objects. Class template greater_ptr<> is an auxiliary predicate for comparing objects given their pointers.

Listing 4. *Destruction Manager.*

dmanager.h :

```
#include <memory>
#include <vector>

using namespace std;

class Destructor;    // forward declaration

class DestructionManager {
  typedef auto_ptr<DestructionManager> DestructionManagerPtr;

  vector<Destructor*> m_destructors;

  static DestructionManagerPtr& get_instance();

  DestructionManager() {}
  ~DestructionManager();

  friend class auto_ptr<DestructionManager>;

public:
  // singleton interface
  static DestructionManager& instance() { return *get_instance(); }

  void register_destructor(Destructor* destructor)
  { m_destructors.push_back(destructor); }

  void destroy_objects();   // destroy the objects
};
```

dmanager.cxx:

```
#include <algorithm>   // for 'sort'
#include <functional>  // for 'greater'

#include "dmanager.h"
#include "destructor.h"

using namespace std;

DestructionManager::~DestructionManager()
```

```
{
  // the Destruction Manager is responsible for managing
  // the memory occupied by Destructor objects
  for (int i = 0; i < m_destructors.size(); ++i)
    delete m_destructors[i];
}

template <class T> class greater_ptr {
public:
  typedef T* T_ptr;

  bool operator()(const T_ptr& lhs, const T_ptr& rhs) const
  { return *lhs > *rhs; }
};

void DestructionManager::destroy_objects()
{
  // sort the destructors in decreasing order
  sort( m_destructors.begin(),
        m_destructors.end(),
        greater_ptr<Destructor>() );

  // destroy the objects
  for (int i = 0; i < m_destructors.size(); ++i)
    m_destructors[i]->destroy();
}
```

Listing 5 gives a sample resource definition (again, insignificant details have been omitted). Function destroy_instance() releases the singleton object from the auto_ptr, and then destroys it. The auto_ptr is no longer the owner of the object, therefore, it would not attempt to delete it when the program terminates. This function is invoked by the TDestructor object, which is hence defined a friend of class Logger. The Logger constructor creates a new TDestructor object, which contains all the information necessary to destroy the Logger object at the right time. The constructor of TDestructor registers it with the Destruction Manager, which ultimately deletes the corresponding singleton, and then deallocates the TDestructor object itself.

Listing 5. *Sample resource definition.*

logger.h :

```
#include <memory>
#include <string>

#include "dphase.h"
#include "destructor.h"

using namespace std;

class Logger {
```

```
  typedef auto_ptr<Logger> LoggerPtr;

  static LoggerPtr& get_instance();
  static void destroy_instance()
  { delete get_instance().release(); }

  Logger();
  ~Logger();

  friend class auto_ptr<Logger>;
  friend class TDestructor<Logger>;

public:
  // singleton interface
  static Logger& instance() { return *get_instance(); }

  void log(string message);
};
```

logger.cxx :

```
#include <iostream>
#include <string>

#include "dphase.h"
#include "destructor.h"
#include "logger.h"

using namespace std;

Logger::Logger()
{
  new TDestructor<Logger>(this, DestructionPhase(1));

  cout << "Logger created" << endl;
}

Logger::~Logger()
{
  cout << "Logger destroyed" << endl;
}

void Logger::log(string message)
{
  cerr << "Logger: " << message << endl;
}
```

The complete example code (available from the C/C++ Users Journal Web site, see above) also defines class Resource, whose destructor uses function Logger::log() to record error messages (if any).

Finally, Listing 6 presents function main().

Listing 6. *Sample main program.*

main.cxx :

```
#include "dmanager.h"
#include "resource.h"
#include "logger.h"

void main(void)
{
  // some code
  Resource::instance().process();

  DestructionManager::instance().destroy_objects();
}
```

Discussion

Instead of using an express notion of destruction phases, an alternative approach could require each singleton to explicitly specify on which other objects it depends. The destruction manager would then perform topological sort of the dependencies graph, thus deducing the phases of destruction automatically. This requires that the constructor of each singleton inform the destruction manager of all the other singletons it depends on. If we pursue a simple solution where the constructor registers pointers to all the other singletons it might use, all those will be created even if some of them are not needed in a particular program execution. A more elaborate solution is possible where singletons are identified by their string names rather than pointers, but this seems like an overkill.

Conclusions

In this paper I examined the existing Singleton realizations, and analyzed their advantages and drawbacks. I also proposed an improved Singleton implementation that detaches the Singleton object from its wrapper, and then used it in combination with other design patterns to achieve certain behavior.

Using the compound Singleton-Registration-Proxy pattern I implemented a Destruction Manager which controls the lifetime of a singleton, and ensures that the given singleton exists as long as it is needed. In fact, it is possible to view the Destruction Manager as a "destructional pattern", as opposed to creational patterns [1] like Abstract Factory, Factory Method or Singleton itself for that matter. These types of patterns lie on the opposite sides of the lifecycle-management spectrum. Certainly, additional research should be conducted (including documenting other known uses) before the Destruction Manager may be pronounced a pattern in its own right.

Among the issues this paper does not address is thread-safety. Each singleton implementation I'm aware of uses variables defined at global scope. In a multithreaded environment this may constitute a problem, should a Singleton have to serve multiple threads. Mutual exclusion in the presence of threads is a fairly well

researched topic, but its application to the Singleton pattern may be non-trivial and thus deserves a detailed examination.

Acknowledgements

The Singleton Destruction Manager was inspired by an example code from [6], where an initialization manager working in phases was used to resolve the mutual initialization order of global variables (without singletons, but rather using two-stage object construction).
The author is thankful to Brad Appleton for interesting discussions that helped to improve this paper. The author is also thankful to Avner Ben and Vitaly Surazhsky for their constructive comments.

References

[1] Gamma, E., et al. "Design Patterns: Elements of Reusable Software Architecture", Addison Wesley, 1995.
[2] Vlissides, J. "Composite Design Patterns", C++ Report, 10(6): 45-47, June 1998.
[3] Meyers, S. "Effective C++", 2nd edition, Addison Wesley, 1998.
[4] Vlissides, J. "Pattern Hatching. Design Patterns Applied", Addison Wesley, 1998.
[5] "Information Technology -- Programming Languages -- C++", International Standard ISO/IEC 14882-1998(E).
[6] Ben-Yehuda, S. "C++ Design Patterns" course, SELA Labs (http://www.sela.co.il).

About the author

Evgeniy Gabrilovich is a Strategic Development Team Leader at Comverse Technology Inc., a developer of multimedia communications processing technology. He holds an M.Sc. in Computer Science from the Technion - Israel Institute of Technology. His interests involve Natural Language Processing, Artificial Intelligence, and Speech Processing. He can be contacted at gabr@acm.org .