An Exception Correct C++ Singleton Template Base Class with Controlled Destruction Order

Peter Nordlund
Email: petern@nada.kth.se, Peter.Nordlund@consultant.com

Technical report CVAP246,
ISRN KTH NA/P--00/14--SE.
Department of Numerical Analysis and Computer Science, KTH (Royal Institute ofTechnology),
S-100 44 Stockholm, Sweden, Oct 2000.

An Exception Correct C++ Singleton Template Base Class with Controlled Destruction Order

Ph. D. Peter Nordlund
CVAP, NADA, KTH Stockholm, Sweden
Email: petern@nada.kth.se, Peter.Nordlund@consultant.com

Abstract

This article describes some problems that may arise when trying to write exception safe C++ code and at the same time using STL. This is examplified with an implementation of a Singleton base class with a Destruction Manager that satisfies Abrahams' Strong Exception Guarantee [Abr97]. The Destruction Manager is based on the "Manager design pattern" by Sommerlad and Buschmann [SB96] and it destroys all instantiated Singletons in a controlled order. One part of the presented code is analyzed for exception safety in a more formal way, as suggested by Sutter [Sut99].

Introduction

The intent of the Singleton Pattern [GHJV95] is to ensure that a class has only one unique instance and to provide a global point of access to that instance. The creation of the Singleton is deferred until the first time it is accessed. The solution in [GHJV95] ignores the problem of destroying the instance before program termination. Motivations for destroying the Singleton could be that it might hold system wide resources, such as e.g. system-scope semaphores or I/O buffers, or just to shut down in an orderly manner, (e.g. closing open files etc.). How to destroy Singletons has been investigated in [Gab99, Gab00, LGS99, Mey98, Vli96]. Solutions relying on the mechanism of static object destruction at program termination are presented in [Mey98, Vli96], but they do not address the destruction order for the Singletons[1]. The destruction order could be important in cases where Singletons use other Singletons (the typical example is a Singleton used for some logging functionality, which should be destroyed as the last object). Moreover, when relying on static object destruction there is no possibility to destroy the Singletons before program termination. (One could imagine a scenario where the Singletons should be destroyed, but not as the last thing to happen in a program execution.) In [Gab99, Gab00, LGS99], the problem with destruction order is addressed, however, none of the above mentioned articles deal with the problem of *how to make a Managed Singleton exception safe* (i.e. make the Managed Singleton to work properly in the case of a thrown exception).

Some serious problems easily arise when you try to write exception safe code, if you intend to use STL containers and algorithms. One big problem is that almost all of the STL algorithms only fulfill the *Basic* Exception Guarantee and none of the STL algorithms fulfill the *Nothrow* Guarantee. A few such problems will be discussed in this article.

There are a number of variations of how to realize the Singleton pattern. In the proposed solution the Singleton is implemented as a template base class that exploits Coplien's "curiously recurring template pattern" [Cop95]. If you do not like this approach it is also easy to use a Singleton adapter class (will also be shown) in the spirit of [SH97, LGS99, VA00]. By using the "Manager Design Pattern" [SB96] and parameterizing the Singleton class on a Manager class, the responsibility of the lifetime of the Singleton can be decoupled from the Singleton class. Since the whole solution is template based, this approach will not impose any performance penalties compared with having the Manager functionality inside the Singleton class. In the original Manager Pattern, the Manager itself is a Singleton. I have simplified this a bit for brevity. The solution presented in this article has been designed with efficiency in mind. The intention being that frequently executed code should not have any performance overhead compared to a Singleton without a Destruction Manager.

[1] The destruction order for static objects in different translation units is not defined in the C++ language.

In contrast to [SH97, Gab99, Gab00, LGS99, VA00] the presented solution has no code concerned with managing functionality of the Singleton (such as e.g. lifetime policies, thread safe locking or allocation policies) inside the Singleton class. All such code is located in the Manager class to get a better separation of concerns and thus promoting ”GPI”[2]. Decoupling the Manager from the Singleton class gives the user of the Singleton class freedom to choose a Manager with arbitrary complexity. He could use a simpler or more complex Manager than the one presented in this article. For examples of simpler managers, see the source code[3]. It could e.g. be a Manager that does not destroy the Singletons even on program termination, if that is preferred for some reason.

After this introduction follows a section about general aspects on exception safety. Then comes an implementation of *”An Exception Correct Singleton Template Base Class with Controlled Destruction Order”* together with an exception safety analysis of the presented code. One part of the code is analyzed according to the ACID approach as suggested by Sutter [Sut99].

Exception Safety

A component may be called exception safe if it behaves ”reasonable” when an exception is thrown. A ”reasonable” behavior would include that no resources are leaked and that the component has a predicable state after the exception has been thrown, so that the program can continue orderly if the exception is caught. Guidelines for how to cope with exceptions can be found in e.g. [Ree96, Str00, Sut97]. A common expectation on a Generic component is that it should be ”exception neutral”, i.e. it should propagate exceptions unchanged to the caller [Str00, Abr99, Sut97].

Definitions of exception safety for Generic Components were first set out by Dave Abrahams [Abr97], see also [Abr99, Str00, Sut97, Sut99]. Operations on such a component may provide one of the following guarantees in the case of an exception, citing [Abr99]:

- The *basic* guarantee: that the invariants of the component are preserved, and no resources are leaked.
- The *strong* guarantee: that the operation has either completed successfully or thrown an exception, leaving the program state exactly as it was before the operation started.
- The *no-throw* guarantee: that the operation will not throw an exception.

Following Reeves [Ree96] notation, I mark potentially exception-throwing operations with the comment `//>x` in the code. Operations fulfilling the Nothrow Guarantee are marked `//NOTHROW`. To fulfill the Strong Guarantee, the ”program state” should not be affected if an exception is thrown. The concept ”program state” is closely related to the concept: ”side effect”. For a more precise analysis of what a ”side effect” and ”program state” is, see [Sut99] (or the sidebar on side-effects). The precise meaning will not make any difference for the discussion of the solution described here. If a ”side effect” happens, the ”program state” will be regarded as having changed. Examples of side effects: When a function is called, a global variable changes its value or a rocket is launched or a file on disk is erased.

Analyzing a Function for Exception Safety

When a function is examined for a certain level of exception safety, every single operation within the function has to be investigated. You could reason about it this way: each operation has a potential return statement immediately after it. You have to investigate every operation to see if it is possible that the function returns (i.e. throws an exception or makes an ordinary return). For each place where

[2] The term GPI was introduced in [VA00]: ”´GPI,´ short for Generic Pattern Implementations. GPI uses C++ templates and generic programming techniques to capture good pattern implementations in an easy-to-use, mixable and matchable form.”

[3] The file `various-managers-test.cpp` at ftp://ftp.nada.kth.se/CVAP/users/petern/singleton-dist-* contains such examples.

it is possible that the function returns, you have to check whether the desired level of exception guarantee is fulfilled. Sutter [Sut99] has formalized this kind of reasoning, which will briefly be described in the next section.

ACID programming

Herb Sutter [Sut99] has suggested a more formal way of analyzing code for exception safety. Later in the article, I will analyze the function `DestructionManager::instantiate` method according to Sutter's method to see if the Strong Guarantee is fulfilled. A very brief summary of the most important concepts follows below.
In short, what Sutter does: He defines a notation for exception safety guarantees inspired from concepts used in the database world. This gives 4 different aspects of exception safety. The aspects are: Atomicity, Consistency, Isolation, and Durability (hence ACID).

For each aspect, 3 consecutive levels are defined. An operation or function call can satisfy either level 1, 2, or 3.
1. No guarantee at all. Designated with a '-'.
2. A statement about effects, if an operation fails. Designated with a lowercase letter, e.g. 'a'.
3. In addition to 2. A statement about effects, if an operation does not fail or cannot fail. Designated with an uppercase letter, e.g. 'A'.

This gives us:
Atomicity:
a If the operation fails, the manipulated object(s) are in their initial state, otherwise the manipulated object(s) are in their final state.
A The operation is guaranteed to succeed and the manipulated object(s) are in their final state.
(Atomicity as defined here only concerns itself with the state of a directly manipulated object.)

Consistency:
c If the operation fails, the system is in a consistent state.
C If the operation fails or succeeds, the system is in a consistent state.

Isolation:
i If the operation fails, there will be no side effects.
I If the operation fails or succeeds, there will be no side effects.

Durability:
d If the operation fails, whatever state was achieved will persist.
D If the operation fails or succeeds, the state persist.

Having these definitions, a functions exception safety level can be described with a 4-tuple, e.g. ACID, AC--, or aCi-.

Using this terminology the Strong Guarantee is equivalent with aCi-. A function is analyzed for aCi-Safety the following way (citing [Sut99]):
1. *[a] For each possible EARLY return, ensure the immediate object is returned to the initial state if the early return is taken.*
2. *[C] For each NORMAL return, ensure the immediate object's state is consistent.*
3. *[i] For each possibly EARLY return, ensure that side effects are undone if the early return is taken.*

Notice that we do not have to regard c-safety since ai-safety implies c-safety.

Before the proposed Singleton class and its exception guarantees are presented, let us have a look at an example discussing some of the problems with STL and exception safety. This example is relevant for the proposed solution.

The Importance of Nothrow Operations

Given that we want to use STL algorithms and containers in an exception safe implementation, a few problems arise. There are no Nothrow insert operations available in STL [Str00, Abr97]. Moreover, (almost all of) the algorithms only fulfill the Basic Guarantee[4] according to the standard. Consider the following scenario:
I want to write a function, A, satisfying the Strong Guarantee. A calls another function, B, that also satisfies the Strong Guarantee. B, however, has *side effects* that cannot be undone if B succeeds. After calling B and while A has not yet returned, I want to take actions, depending on the outcome of the call to B (e.g. different actions whether the call to B succeeded or not). This means that I cannot reorder my operations to do e.g. `insert` of the return value from B into a sorted `vector` before I have called B since I don't have the return value yet, I *must* do it afterwards. Now I'm in trouble since I have no Nothrow `insert`[5]..... From now on *all* of the operations in the rest of function A must be Nothrow if A shall satisfy the Strong Guarantee.

A major problem is that there are quite a lot of operations in STL that are not Nothrow, `insert` is only one example. Take `find_if` as another example. Even if `find_if` is provided with a nonthrowing predicate, the Standard does not guarantee success. Below is a commonly used implementation of `find_if` (from [SGI]).

```
template <class _InputIter, class _Predicate>
inline _InputIter find_if(_InputIter __first, _InputIter __last,
                          _Predicate __pred,
                          input_iterator_tag)
{
  while (__first != __last && !__pred(*__first))
    ++__first;
  return __first;
}
```

This implementation *is* actually a Nothrow algorithm, given that we use standard containers and valid iterator ranges and a Nothrow predicate, (STL iterators on a valid range will never throw [Str00]) still the C++ standard only promises the Basic Guarantee. So if we want to write portable Nothrow code in this example, we have to make our own copy of SGI's actual implementation, (which I find quite unfortunate).

The Suggested Singleton Implementation

The `Singleton` class (see listing 1) is parameterized on the type `MANAGER`, which is a class responsible for the lifetime of the Singleton objects:

[4] `uninitialized_fill`, `uninitialized_fill_n` and `uninitialized_copy` provides the Strong Guarantee.

[5] Although the Standard does not require this, it is possible to make `vector::insert` of single elements conditionally Nothrow, given that the element-type has its constructors and assignment-operator Nothrow and, for a `vector V`, `V.capacity() > V.size()`. See [Str00] for an extensive example of a vector implementation.

```
// Forward declarations.
template <int D_PHASE = 1 /* Destruction phase */ > class DestructionManager;
typedef DestructionManager<> DefaultManagerType;

template <typename T, typename MANAGER = DefaultManagerType>
class Singleton {
public:
  typedef MANAGER ManagerType;
  typedef T       SingletonType;
  inline static T& instance() {
    T* typeArg;
    return MANAGER::instance(typeArg); //>x
  }
  inline static const T& constInstance() { return instance(); } //>x

protected:
  Singleton()  { };                        // NOTHROW
  ~Singleton() { };                        // NOTHROW
private:
  Singleton(const Singleton&);             // Shouldn't be used
  Singleton& operator=(const Singleton&); // Shouldn't be used
};

// Singleton Adapter class. Will "convert" a class T into a Singleton.
template <typename T, typename MGR>
struct SingletonAdapter: public Singleton<T, MGR> { };
```

Listing 1.

Note that the `Singleton` class has no virtual functions, so there is no need for a vtable for the class. The fact that this base class has no virtual destructor implies that we should not delete any objects via a `Singleton` pointer, (which is not possible anyway, since the destructor is protected).

Destroying Singletons

The class `DestructionManager` is one possible implementation of a `MANAGER`. Following the solution by Gabrilovich [Gab99, Gab00] the Singleton is assigned a "Destruction Phase", i.e. the lower Destruction Phase value, the later the Singleton will be destroyed. Singletons with equal Destruction Phase will be destroyed in the opposite order of registration[6]. One could imagine more complicated strategies for destruction order but this strategy will serve its purpose in most contexts. Before getting into details, I begin with a code excerpt showing how the Managed Singleton can be used in a program (see listing 2):

```
...
// Coplien's curiously recurring template pattern [Cop95].
class Test: public Singleton<Test> {
  friend class InstanceManager<Test>; // Will call ctor and dtor.
...
private:
  ~Test() throw() { ... }
   Test() { ... }
};

int main(int argc, const char** argv) {
  try {
    DestructionManager<>::AutoDestructor autoDtor;
    Test::instance();
    ...
  } catch (...) { std::cerr << "Caught ..." << std::endl; throw; }
  return 0;
}
```

Listing 2.

[6] I have chosen to register the Singletons immediately after their construction. It would also be possible to register them immediately before construction, more on that later.

When `Test::instance()` is called for the first time the `Test` object is created. If a program is terminated normally or via an exception does not matter, when the `DestructionManager<>::AutoDestructor` goes out of scope all `Singleton` objects created so far are deleted[7]. (As already mentioned they are deleted in a particular order but we will get back to that later).

The Destruction Manager will in some way maintain a collection of references to all Singletons created. All these Singletons will have different types. How shall we manage to handle different types of Singletons in one Destruction Manager? An easy approach would be to have a common Singleton base class:

```
class SingletonBase {};
template <typename T, typename MANAGER> class Singleton: public SingletonBase
```

But it is not necessary to mess up the `Singleton` class with implementation details of the Destruction Manager. Instead the Destruction Manager could have a helper class, `InstanceManager<T, MANAGER>`. `InstanceManager` has a base class `InstanceManagerBase` (see listing 3). Via this base-class it is possible to handle a collection of `Singletons`. Moreover, the `InstanceManager` class is using the ”Resource acquisition is initialization” idiom [Str97, Str00] and encapsulates the dynamic allocation of the `Singleton`, thus reducing the need for try blocks when constructing the `Singleton`. In the suggested implementation, `InstanceManager` is a template class that is specialized for a particular Manager class. By specializing the `InstanceManager` class for a particular type `T`, a deviating behavior can be obtained for one or more Singleton classes, although only one Manager type is used. An example of such a deviating behavior where one particular Singleton class reads data from disk when it is instantiated can be found in the source code[8].

```
// Master template, should never be used.
template <typename T, typename MANAGER = DefaultManagerType>
class InstanceManager{};

// Base-class.
struct InstanceManagerBase { virtual ~InstanceManagerBase() {}; /*NOTHROW*/};

// Partly specialized InstanceManager
//
template <typename T, int D_PHASE /* Destruction phase */>
class InstanceManager<T, DestructionManager<D_PHASE> >:
  public InstanceManagerBase {
  friend class DestructionManager<D_PHASE>;
public:
  inline static bool isInstantiated() { return 0 != theInstance_; }// NOTHROW
private:
  inline static T& get() { return *theInstance_; }    // NOTHROW
  InstanceManager() { theInstance_ = new T; }         //>x;
  virtual ~InstanceManager() {
    delete theInstance_;
    theInstance_ = 0;                                 // NOTHROW
  }
  // Data member
  static T* theInstance_;  // Points to the Singleton when !=0.
};
```

Listing 3.

[7] If no `AutoDestructor` goes out of scope and the program is terminated normally via a return from `main()`, the Singletons will still be deleted, since there is a `static AutoDestructor` member in the `DestrucionManager` class that will ensure proper cleanup after a normal exit from `main()`.

[8] The file `various-managers-test.cpp` at ftp://ftp.nada.kth.se/CVAP/users/petern/singleton-dist-* contains such an example.

Now we can maintain a register of `Singletons`, implemented as a collection of pointers to `InstanceManagerBases`. In the `DestructionManager::instantiate` method, which will be shown later, the `InstanceManager` pointers are inserted in a STL `list` where they later are sorted due to their Destruction Phase. When the `DestructionManager::AutoDestructor` is destructed this `list` will be iterated over and all registered Singletons will be deleted.

Requirements on the Class that Should be Made into a Singleton:

It is quite common that some requirements are put up on the client class of a generic component. For example in STL, all types that instantiates a container must have a nonthrowing destructor. Now, if we have an originally non-singleton class `T` that is made into a Singleton by inheriting in the following way: `class T: public Singleton<T> {...};`
what are then the requirements on `T` ? Here `T` has to be a class that in itself fulfills all kinds of exception guarantees that you want it to (at least the Basic Guarantee). We will focus on the parts that interact with the `Singleton` base class and that we would like to fulfill the Strong Guarantee. Then `T`'s constructor also has to fulfill the Strong Guarantee. Why this is both necessary and sufficient will be seen when the `DestructionManager::instantiate` function is examined for exception safety in the next section. `T` should also have a non-throwing destructor[9].

Suggested Destruction Manager Satisfying the Strong Exception Guarantee

What should be achieved in short, is the following:

- Singletons should be constructed and registered in a collection of Singleton-references of some kind.
- When the Singletons should be destroyed, this collection must have been sorted on Destruction Phase. The sorting should be stable, that is, the relative order of equivalent elements should be preserved, so that Singletons with equal Destruction Phase are destructed in opposite order of registration.

A central question when writing exception safe code is which order operations should be performed in. Since the construction of a (subclassed) Singleton may cause side effects that are not possible to undo, the registration of the Singleton may not fail, if it is performed *after* the Singleton has been constructed. The sorting may also not fail, if the sorting is done *after* the construction of the Singleton. This leaves us with a few alternatives. One could begin to register the Singleton and sort the collection of Singleton references *before* the Singleton is constructed. If the Singleton construction fails, the registration could be rolled back. If the collection is sorted *before* Singleton construction, it is easy to cope with exceptions thrown during the sorting, since again, it is trivial to rollback the registration so that program state is unchanged.

I intend to use containers and algorithms from the Standard Library (STL) to hold the collection of Singleton references. As already mentioned there are no Nothrow insert operations available in STL [Str00, Abr97], so if we want to register *after* Singleton creation, the only possibility is to use a `list` as the container and the `list::splice` member function which is Nothrow to add an element from another (already created) `list` to the `list` holding the register. Moreover, the `list` class has a conditionally Nothrow `sort` function. This suits us fine! `list::sort` will be Nothrow given that the sort's comparison function is Nothrow. If we chose to register *before* Singleton creation, we can use `vector` or `deque`, but then we also have to maintain the collection sorted *before* Singleton creation. We cannot delay the sorting until the Singletons should be destructed since there is no STL `sort`

[9] One reason to have a nonthrowing destructor is that if an exception is thrown from a destructor when the destructor is called during stack unwinding caused by another exception, `terminate()` will be called. See also [Ree96, Sut97, Str00, Abr99] for arguments against throwing destructors. A good example of this approach to non-throwing destructors is the C++ standard, which says: "No destructor operation defined in the C++ Standard Library will throw an exception." It is also illegal to instantiate a standard container with a type whose destructor throws exceptions.)

algorithm available for `vector` or `deque` that has a Nothrow guarantee. We cannot even use other STL algorithms to do anything clever to get the collection sorted, since as already mentioned (almost all of) the algorithms only fulfill the Basic Guarantee. If we register *before* Singleton creation a `multimap` could also be used. Then we could get the collection sorted automatically on insertion.

I feel that the most natural solution is to register the Singleton immediately *after* construction, so there is no other choice than the `list` alternative.

Investigating the Suggested Solution for Exception Safety

The only protected functions available are the Singleton's constructor and destructor and they do nothing so they are Nothrow. The only publicly available methods in the suggested solution are `Singleton::instance()`, `DestructionManager::AutoDestructor()` and `DestructionManager::~AutoDestructor()`, so it suffices if those methods are found to be exception safe, which will be examined below, after the listing of the `DestructionManager` class[10] (see listing 4):

```
class DestructionManagerBase {
protected:
  class AutoDestructor; // Use this to get automatic Singleton destruction.
  friend class AutoDestructor;

  // Exception classes
  ...
  typedef std::pair<int, InstanceManagerBase*> SingletonCollElementType;
  typedef std::list<SingletonCollElementType>  SingletonCollectionType;

  static void Register(SingletonCollectionType& newSingleton); // NOTHROW

  // Data member
  static SingletonCollectionType singletons_;
private:
  static void destruct() throw(DestructOngoing);    //>x
  DestructionManagerBase();// Prevent obj. creation.

  // To get automatic cleanup at normal program termination even
  // if no AutoDestructor is instantiated in main().
  static AutoDestructor autoDestructor_; // Data member
};

template <int D_PHASE = 1 /* Destruction Phase */>
class DestructionManager: protected DestructionManagerBase {
  template<typename T, typename MANAGER> friend class Singleton;
  typedef DestructionManager<D_PHASE>  SelfType_;
protected:
  typedef DestructionManagerBase       BaseType;
public:
  enum { DestructionPhase = D_PHASE }; // For compile-time computations.

  typedef BaseType::AutoDestructor    AutoDestructor;

  // Exception classes
  typedef BaseType::CircularInstantiation CircularInstantiation;
  ...
private:
  template <typename T> inline static T& instance(T*); //>x
  template <typename T> static void instantiate();   //>x
};
```

Listing 4.

[10] In the original Manager Pattern the Manager is a Singleton. I have simplified this a bit for brevity. Instead of having a method instantiating the one and only object, I have a static variable `singletons_` for the collection of Singletons. I have also prohibited the possibility to construct a `DestructionManager` object.

Some of the code for the `DestructionManager` is put in the base class `DestructionManagerBase` to minimize the number of template functions and classes instantiated and thus reducing code bloat. Having two `DestructionManagers` with different template parameters `D_PHASE` will result in two different classes, but since they both share data members and functions from the `DestructionManagerBase` class they will more or less conceptually act as if they were one single class. The only thing that will differ is that different `D_PHASE` parameters will be used in a few function calls.

We start to examine `Singleton::instance()`:
We want to keep the state of both the Singleton and Destruction Manager unchanged if an exception is thrown, so we have to think of *the order* things are performed in, especially when to assign the `InstanceManager::theInstance_` member, since then the program state is changed. To avoid problems with dangling pointers after the Singletons have been destroyed, a Singleton should always be accessed via the `Singleton::instance()` method and references to it should never be kept from time to time.

```
template <typename T, typename MANAGER>
inline T& Singleton<T, MANAGER>::instance() {return MANAGER::instance<T>();} //>x
```

The `DestructionManager::instance` function (listing 5) contains two ”Locks”. The template class `Lock` simply throws an exception if the class invariant, number of objects alive at the same time <= 1, is violated. I.e. when the caller tries to create a second `Lock` object, the constructor throws. The first lock will throw an exception if circular calls to `DestructionManagerBase::instance` are detected. Such circular calls may arise if the constructor of one Singleton calls another Singleton’s `instance()` method, which in turn calls the first Singleton’s `instance()` method[11]. The second lock will throw an exception if the Destruction Manager has started to destroy the registered Singletons and any of the Singleton’s destructors tries to call an already destroyed, or not yet created, Singletons `instance()` method.

```
template<int D_PHASE>
template <typename T>
  inline T& DestructionManager<D_PHASE>::instance(T*) {
  if (!InstanceManager<T, SelfType_>::isInstantiated()) { // NOTHROW

    // Here we could acquire a lock if we want to use "Double-Checked Locking"
    // for thread-safe creation of Singleton (details omitted).

    Lock<CircularInstantiation, T> throwIfCircularInstantiation;  //>x
    try { Lock<DestructionOngoing> throwIfDestructionOngoing; }
    catch(...) { throw InstantiationFailedDestructionOngoing(); } //>x

    instantiate<T>();                           //>x
  }
  return InstanceManager<T, SelfType_>::get();  // NOTHROW
}
```

Listing 5.

Notice that all locking operations are performed inside the if-clause so they should thus be executed very infrequently and will add a negligible performance overhead. If any of the locks throws an

[11] Since the first lock takes both an exception class and the subclassed Singleton type as template parameters there will exist one locking class per subclassed Singleton class. There can be several simultaneously calls to the template-function `DestructionManagerBase::instance`, an exception is not thrown until one particular subclassed Singleton’s `instance()` function is called a second time while the first call has not yet completed.

exception, program state has not yet changed, so until calling the `DestructionManager::instantiate` method (see listing 6) the Strong Guarantee is fulfilled.

Now we will check the `instantiate` method:

```
template<int D_PHASE>
template <typename T>
void DestructionManager<D_PHASE>::instantiate() {

  // Pre-allocating a list for splicing.
  SingletonCollectionType
    preAlloc(1, SingletonCollElementType(D_PHASE, 0));        //>x0
  preAlloc.back().second = new InstanceManager<T, SelfType_>; //>x1
  BaseType::Register(preAlloc);                               // NOTHROW
}

void DestructionManagerBase::Register(SingletonCollectionType& newSingleton) {
  singletons_.splice(singletons_.end(), newSingleton);  // NOTHROW
}
```

Listing 6.

If the construction of the local variable `preAlloc` throws an exception, program state has not changed yet. If `new InstanceManager<T, SelfType_>` throws, program state has not yet changed either and the rules of the C++ language ensures that the partially created object is properly deallocated, so no resources have leaked either. After `new InstanceManager<T, SelfType_>` has succeeded, program state has changed, since the static variable `InstanceManager::theInstance_` now has changed value. (It would be easy to roll back the assignment of `InstanceManager::theInstance_`, but the creation of the subclassed Singleton may have caused side effects that are not possible to undo.) So from now on no operations may fail. As the last operation `DestructionManagerBase::Register` is called. `list::splice` is a Nothrow operation which will never fail.

In the last section we required that the subclassed Singletons constructor should satisfy the Strong Guarantee. This is a necessary requirement, for if the constructor has a side effect that can not be undone and throws after that, there is no way that we can make the operation `new InstanceManager<T, SelfType_>` to fulfill the Strong Guarantee, which is necessary if the `instantiate` function should satisfy the Strong Guarantee. A general observation can be made here: To write a function satisfying the Strong Guarantee, all function calls within that function must, at least, fulfill the Strong Guarantee *if they have any side effects that can not be undone*. This is a necessary but not sufficient condition.

The Strong Guarantee says nothing of the ability to undo a successful operation or not to affect program state on success, so after a successful Singleton subclass object creation, all operations performed in the rest of the function must satisfy the Nothrow Guarantee (which they do in this case).

`DestructionManager::AutoDestructor()` is trivial, it does nothing so it satisfies the Strong Guarantee. Finally we examine `DestructionManager::~AutoDestructor()`(see listing 7):

```
DestructionManagerBase::~AutoDestructor() throw(DestructOngoing) { destruct(); }

// Recursive calls to destruct will throw,
// but it's not possible for a client to achieve such a recursion,
// unless an AutoDestructor is instantiated in a Singleton dtor,
// which is plain WRONG to do!!
//
void DestructionManagerBase::destruct() throw(DestructOngoing) {

  // Trying to create Singletons from now is an error!
  Lock<DestructionOngoing> throwIfDestructionOngoing;  //>x

  // select1st is a SGI nostadard func. adapter, see [SGI].
  typedef select1st<SingletonCollElementType> s1T;
  typedef greater_equal<int> geqT;

  // Stable sort on destruction phase.
  // boost::compose_f_gx_hy[12] is a nostandard func adapter, see [Jos99].
  singletons_.sort(boost::compose_f_gx_hy(geqT(), s1T(), s1T())); // NOTHROW

  try {
    // begin(), !=, end(), ++it are all NOTHROW
    for(SingletonCollectionType::const_iterator it = singletons_.begin();
        it != singletons_.end(); ++it)
      delete (*it).second; // If this operation throws we are in trouble!
  } catch(...) { terminate(); }
  singletons_.clear();     // NOTHROW
}
```

Listing 7.

Here we can notice an implication of a destructor that may throw. The `Lock` in `destruct()` should not throw, unless a serious programmer error has been made (see comments in the code above). If this possible exception is not allowed to propagate out from `~AutoDestructor()`, we have violated the ”exception neutrality”. Since this exception should never happen (or at least it should be caught in the destructor that caused the exception), I leave it as it is.

The comparison object used is Nothrow. This implies that `list::sort()` also is Nothrow [Str00].

No STL iterators will throw if they are used on a proper range [Str00]. Each time `delete` is called, an `InstanceManager` is deleted. The `InstanceManager` will in turn delete the `Singleton` it owns. As already mentioned, we require the subclassed `Singleton`s destructors to be Nothrow, so `delete` should not throw[13]. This makes the loop Nothrow.

Finally `list::clear()` is called. This is a Nothrow operation.

---

[12]
```
// For those not familiar with function adapters,
// something like the code below could be used instead.
// Comparison functor
struct DestructionManagerBase::PhaseGreaterEqual:
  public std::binary_function<SingletonCollElementType, SingletonCollElementType, bool> {
  bool operator() throw()
    (const SingletonCollElementType& x, const SingletonCollElementType& y) const {
    return x.first >= y.first;
  }
};
singletons_.sort(PhaseGreaterEqual()); // NOTHROW
```

[13] If `delete` for some reason should throw anyway, then we do not know how many Singletons that were deleted before the exception was thrown, (even if we bother to find out, the situation cannot be handled because there is no appropriate action to take) so here I follow an advice by Reeves [Ree96] *”If you get stuck, call terminate()”*. For an elaboration of this problem, see [Sut97]. Getting out of this situation by calling `terminate()` means that we violate the ”exception neutrality”, but there is not much to do about that.

Investigating a Function using the ACID Approach

Below follows once again, an analysis of `DestructionManager::instantiate`, this time according to the 3 steps for aCi-safety [Sut99]:

1. *[a] For each possible EARLY return, ensure the immediate object is returned to the initial state if the early return is taken.*
2. *[C] For each NORMAL return, ensure the immediate object's state is consistent.*
3. *[i] For each possibly EARLY return, ensure that side effects are undone if the early return is taken.*

This is basically the same analysis as performed earlier in the article, but below it is done in a more formalized way:

```
template<int D_PHASE> template <typename T>
void DestructionManager<D_PHASE>::instantiate() {
  // Pre-allocating a list for splicing.
  SingletonCollectionType
    preAlloc(1, SingletonCollElementType(D_PHASE, 0));        //>x0
  preAlloc.back().second = new InstanceManager<T, SelfType_>; //>x1
  BaseType::Register(preAlloc);                               // NOTHROW
}
```

1. If early return is taken at `x0` or `x1`, the immediate object is still in its initial state, since at both `x0` and `x1` only a local variable is manipulated.
2. There is only one normal return. At this point `singletons_` has got a new entry and this entry contains a pointer to a valid object that in turn has a valid pointer to a `Singleton`, so the immediate object is in a consistent state.
3. If early return is taken at `x0` or `x1` no side effects have yet occurred. It is important to notice that if an early return is taken at `x1` no side effects will occur due to the subclassed `Singleton`'s constructor taking an early return, since there was a requirement put up on the subclass that its constructor should fulfill aCi-safeness itself.

Exception Specifications

To complete this Managed Singleton code we have to think of exception specifications. Should we have any? I would say no, not for the `Singleton` class, since we do not know which classes that will use it as a base class. Putting up exception specifications will limit the usability of it, since then the subclass cannot be allowed to throw anything else than the `Singleton` class itself may throw. For elaboration on this topic, see [Sut97].

Making Singleton Creation Thread-safe

The `AutoDestructor´s` destructor is not thread-safe, so threads should be joined before executing the destructor. I have put comments in the `DestructionManager::instance` method where ”Double-Checked Locking” [SH97, Vli96] should appear to make Singleton creation thread-safe. The lock should then, of course, be acquired with a Guard object that will release the lock in its destructor as proposed in [SH97], so that no resources are leaked if an exception is thrown after the lock has been acquired. (It would be quite handy to have the `DestructionManager` class to take a template parameter specifying the locking policy, but I have not implemented that feature here.)

Summary

Some problems of using STL when requiring Nothrow operations have been discussed. A Singleton with a Destruction Manager exploiting the ”Manager pattern” [SB96] has been presented. The Destruction Manager destroys all created Singletons in a controlled order. The Singleton and the Destruction Manager have both been shown to satisfy the Strong Exception Guarantee. The Destruction Manager will detect circular calls to Singleton instantiation. The Singletons do not necessarily have to be destroyed at program termination, they may be destroyed earlier if desired.

The source-code for this article can be obtained from
ftp://ftp.nada.kth.se/CVAP/users/petern/singleton-dist-*

References:

[Abr97] Abrahams, D., ”Exception Safety in STLport”, first published in 1997, at http://www.ipmce.su/~fbp/stl, can now be found at http://www.stlport.org/doc/exception_safety.html

[Abr99] Abrahams, D. ”Exception-Safety in Generic Components”, Generic Programming: proceedings of a Dagstuhl Seminar, April 27-May 1, 1998, Wadern, Germany, Jazayeri M., Loos, R. and Musser, D., eds., Springer Verlag, 1999.

[Jos99] Josuttis, N. ”Compose library”, http://www.boost.org/libs/compose/index.htm, June 1999.

[Cop95] Coplien, J., ”Column without a name” C++ Report Feb. 1995.

[Gab99] Gabrilovich, E., ”Controlling the Destruction Order of Singleton Objects”, C/C++ Users Journal, Oct. 1999.

[Gab00] Gabrilovich, E., ”Destruction-Managed Singleton: a Compound Pattern for Reliable Deallocation of Singletons”, C++ Report, Mar 2000.

[GHJV95] Gamma, E., R. Helm, R. Johnson and J. Vlissides, ”Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.

[LGS99] Levine, D. L., C. D. Gill and D. Schmidt, ”Object Lifecycle Manager”, Patter Languages of Programming Conference, Allerton Park, Illinois, USA Aug 1999, see also http://www.cs.wustl.edu/~schmidt/ObjMan.ps.gz for a later version.

[Mey98] Meyers, S., ”Effective C++”, 2nd ed, Addison-Wesley, Reading, MA, 1998.

[Ree96] Reeves, J., ”Coping with Exceptions”, C++ Report, March 1996, see also http://meyerscd.awl.com/DEMO/MAGAZINE/RE_FRAME.HTM

[SH97] Schmidt, D. C. and T. Harrison ”Double-Checked Locking - An Object Behavioral Pattern for Initializing and Accessing Thread-safe Objects Efficiently” in Pattern Languages of Program Design (R. Martin, F. Bushmann, and D. Riehle, eds.), Addison-Wesley, Reading, MA, 1997, see also http://www.cs.wustl.edu/~schmidt/DC-Locking.ps.gz

[SGI] SGI’s ”Standard Template Library Programmer's Guide”, http://www.sgi.com/Technology/STL/ version 3.2, April 1999.

[SB96] Sommerlad, P. and F. Buschmann, ”The Manager Design Pattern” in proc. of the 3rd Pattern Language of Programming conf, Sept 1996, see also http://siesta.cs.wustl.edu/~schmidt/PLoP-96/sommerlad.ps.gz

[Str97] Stroustrup, B. ”The C++ programming language” 3rd ed., Addison-Wesley, Reading, MA, 1997.

[Str00] Stroustrup, B. ”The C++ programming language , Special Edition”, Addison-Wesley, Reading, MA, 2000. Appendix E ”Standard-Library Exception Safety” See also draft at http://www.research.att.com/ ~bs/safe0.html.

[Sut97] Sutter, H. ”Exception-Safe Generic Containers” C++ Report, Sept and Nov-Dec 1997, for an updated version, see http://meyerscd.awl.com/DEMO/MAGAZINE/SU_FRAME.HTM

[Sut99] Sutter, H. ”Guru of the week #61 ‘ACID Programming’ ” http://www.peerdirect.com/Resources/gotw061a.html, 1999.

[Vli96] Vlissides, J., ”To Kill a Singleton” C++ Report, vol 8 pp 10-19, June 1996.

[VA00] Vlissides, J. and A. Alexandrescu, ”To Code or Not to Code, Part 1” C++ Report, March 2000.

Sidebar on Side Effects

The precise meaning of what a side effect is, seems to be a bit hazy. Opinions differ among people. But, below are two excerpts from [Sut99] which I agree with:
”Local effects include all effects on the visible state of the immediate object(s) being manipulated. Here the visible state is defined as the state visible through member functions as well as free functions (e.g., operator<<()) that form part of the interface.”
”Side effects include all other changes, specifically changes to other program states and structures as well as any changes caused outside the program (such as invalidating an iterator into a manipulated container, writing to a disk file, or launching a rocket), that affect the system's business rules, as they are expressed in object and system invariants. So, for example, a counter of the number of times a certain function is called should normally not be considered a side effect on the state of the system if the counter is simply a debugging aid; it would be a side effect on the state of the system if it recorded something like the number of transactions made in a bank account, because that affects business rules or program invariants.”

Another example of a side effect: You have a function containing a static object `Throw9timesOutOf10` that throws an exception 9 times out of 10 randomly, that is all that object does. (I do not know what you should use such a `Throw9timesOutOf10` for, but for the sake of the discussion.)

```
f() {
  static Throw9timesOutOf10 randomThrower;
  // Do some other stuff that affect the business rules.
  f2(); // May throw.
}
```

Now if `randomThrower` does not throw the first time `f()` is called AND `f2()` throws. Has program state changed? -Yes it has. Since from now on, when `f()` is called, it is guaranteed that `randomThrower` will not throw, since static variables are only initialized once during program execution. Program state has changed, because now you are guaranteed to pass the first operation in `f()`. Before, you only had one chance out of ten to pass the first operation. This example also points out that static variables have a totally different characteristic compared to automatic variables when it comes to exception analysis.