13. Generalized Functors

```
#include <iostream>
#include <functional>
#include <string>
#include <vector>
#include <algorithm>

class status {
   std::string name_;  bool ok_;
public:
   status(const std::string& name): name_(name), ok_(true) {}
   void break_it() { ok_=false; }
   bool is_broken() const { return !ok_; }
   void report() { std::cout<<name_<<" is "<<
      (ok_ ? "working normally":"terribly broken")<<std::endl;
    }
};
```

13. Generalized Functors

```
int main(){
   std::vector<status> stats;
   stats.push_back(status("status_1"));
 … 2, 3, 4
   stats[1].break_it(); stats[3].break_it();
   // 1: bad multiple end calls
   for(std::vector<status>::iterator it=stats.begin();
        it!=stats.end(); ++it)  it->report();
   // 2: better with for_each, but how?
   std::for_each(stats.begin(), stats.end(),
        std::mem_fun_ref(&status::report) ); // ???
   // 3: even better & clearer & easier to remeber
   namespace PH = std::placeholders;
   std::for_each(stats.begin(), stats.end(),
        std::bind(&status::report, PH::_1) );
}
```

13. Generalized Functors

```
int main(){ // context as before
   std::vector<status*> stats;
   … 2, 3, 4
   stats[1]->break_it();   stats[3]->break_it();

   std::for_each(stats.begin(), stats.end(),
       std::mem_fun(&status::report) );
   // not compiling with std::mem_fun_ref(&status::report)

   // but still ok with the same bind
   namespace PH = std::placeholders;
   std::for_each(stats.begin(), stats.end(),
       std::bind(&status::report, PH::_1) );
}
```

13. Generalized Functors

```
int main(){// context as before
   typedef std::shared_ptr<status> Sptr;
   std::vector<Sptr> stats;
   stats.push_back(Sptr(new status("status_1")));
   … 2, 3, 4

   stats[1]->break_it(); stats[3]->break_it();

   // no way with std::mem_fun_ref and std::_mem_fun
   // but still ok with the same bind
   namespace PH = std::placeholders;
   std::for_each(stats.begin(), stats.end(),
       std::bind(&status::report, PH::_1) );
}
```

13. Generalized Functors

ISO/IEC JTC1 SC22 WG21 N3092

Date: 2010-03-26

ISO/IEC IS 14882

ISO/IEC JTC1 SC22

```
// functional:
namespace std {
…
  template<class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
  template<class R, class F, class... BoundArgs>
    unspecified bind(F&&, BoundArgs&&...);
…
} // then, how to store a bound command ?
```

13. Generalized Functors

```
#include <iostream>
#include <functional>

class base {
public: virtual void print() { std::cout<<"I am base.\n“; } };

class derived: public base {
public: virtual void print() {std::cout<<"I am derived.\n"; } };

int main() {
   base b;   derived d;
   auto f = std::bind(&base::print, std::placeholders::_1);
   f(b); f(d);
   base* p = &b;       f(*p);
         p = &d;       f(*p);
}
```

13. Generalized Functors

```
#include <iostream>
#include <functional>

class base { /* as above */ };
class derived : public base { /* as above */ };

template <class Bound>
class F {
public:
   void operator()(base& b) {
        bound_(b);
   }
   F(Bound&& bind): bound_(bind){} // move it !
private:
   Bound bound_;
};
```

13. Generalized Functors

```
// and now instantiating with the unknown:

int main() {
   base b;
   derived d;
   F<decltype(bind(&base::print, std::placeholders::_1))>
            f(bind(&base::print, std::placeholders::_1));
   f(b);
   f(d);

   base* p = &b; f(*p);
         p = &d; f(*p);
}
```

14. Singletons

Static Data + Static Functions != Singleton

```
class Font { … }; // in Font.h
class PrinterPort { … }; // in PrinterPort.h
class PrintJob { … }; // in PrintJob.h

class MyOneAndOnlyPrinter {
public: static void AddPrintJob(PrintJob& job) {
             if( (printQueue_.empty() && PrintPort_.avail())
                 printingPort_send(job.Data());
             else
                 printQueue_.push(job);
         }
private:
         static std::queue<PrintJob> printQueue_;
         static PrinterPort printingPort_;
         static Font defaultFont_;
};
```

14. Singletons

```
PrintJob somePrintJob("MyDocument.txt");
MyOneAndOnlyPrinter::AddPrintJob(somePrintJob);
```

Problem 1: static == ~virtual : Erweiterung/Modifikation nur durch Quelltexteingriff

Problem 2: keine Kontrolle über Initialisierung und *Cleanup*

What if defaultFont_ depends on printingPort_.speed()?

C++ does NOT define the order of initialization of static objects found in different translation units :-!

„Singleton implementations therefore concentrate on creating and managing a unique object while not allowing the creation of another one."

14. Singletons

The basic idiom:

```
// Header-Datei Singleton.h
class Singleton
{
public:
   static Singleton* Instance() // Einziger Zugriffspunkt
   {
      if (!pInstance_) // lazy creation
         pInstance_ = new Singleton;
      return pInstance_;
   }
   ... operations ...
private:
   Singleton(); // Erzeugen neuer Singleton-Objekte verhindern
   Singleton(const Singleton&); // Kopieren verhindern
   static Singleton* pInstance_; // Die einzige Singleton-Instanz
};
// Implementierungsdatei Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
```

14. Singletons

Geht‘s auch einfacher?

```
// Header-Datei Singleton.h
class Singleton
{
public:
   static Singleton* Instance() // Einziger Zugriffspunkt
   {
      return &instance_;
   }
   int DoSomething();
private:
   static Singleton instance_;
};
// Implementierungsdatei Singleton.cpp
Singleton Singleton::instance_;
```

14. Singletons

Problem:

C++ does NOT define the order of initialization for statically initialized objects found in different translation units ☹

```
// SomeFile.cpp
#include "Singleton.h"
int global = Singleton::Instance()->DoSomething();
```

14. Singletons

Ein paar kosmetische(?) Verbesserungen:

```
class Singleton
{
   static Singleton& Instance(); // warum besser?
   ... operations ...
private:
   Singleton();
   Singleton(const Singleton&);
   Singleton& operator=(const Singleton&);
   ~Singleton(); // warum besser?
};
```

14. Singletons

Lazy creation sorgt dafür, dass Singletons nur bei Bedarf erzeugt werden. Sie werden aber nie beseitigt!

Ist jedes Singleton ein *memory leak*?

NEIN! ***(Memory leaks appear when you allocate accumulating memory and lose all references to it.)***

Aber potenziell ein *resource leak* ☹

14. Singletons

„The only correct way to avoid resource leaks is to delete the Singleton object during the applications shutdown.“

```
// as with Scott Meyers‘:
Singleton& Singleton::Instance()
{
   static Singleton obj;
   return obj; // who cares for its destruction?
}
     int std::atexit(void (*function)(void));
     // register a function to be called at normal
     // process termination
```

14. Singletons

```
int std::atexit(void (*function)(void));
```

Hatte ein ernsthaftes Problem in der Vergangenheit, welches sowohl im C- wie auch im C++ -Standard nicht geklärt war:

Functions so registered are called in the reverse order of their registration. **– klingt vernünftig – ist aber widersprüchlich:**

```
#include <iostream>
#include <cstdlib>

void f() {std::cout<<"Programm Exit\n";}
void bar() {std::cout<<"bar()\n";}
void foo() {
    std::cout<<"foo()\n";
    std::atexit(bar);
}

int main() {
    std::atexit(f);
    std::atexit(foo);
    std::exit(-1);
}
```

Inzwischen behoben:

```
foo()
bar()
ProgrammExit
```

14. Singletons

Dennoch bleibt ein wichtiges praktisches Problem (The dead Reference problem):

Eine Applikation habe (u.a.) drei Singletons für
`Keyboard` – die EINE Tastatur
`Display` – der EINE Bildschirm
`Log` – das EINE Protokollmedium (File, 2. Konsole, LCD-Display, ...)

Gut: `Log` wird überhaupt nur im Fehlerfall erzeugt ☺

Aber folgendes Szenario (hier *KDL* genannt):

1. **`Keyboard` erfolgreich konstruiert.**
2. **`Display` Konstruktion scheitert -> Erzeugt `Log` und protokolliert den Fehler**
3. **`std::exit`: `Log` vernichtet vor `Keyboard`**
4. **`Keyboard` Destruktion scheitert und will das im Protokoll vermerken ☹**

14. Singletons

„The program steps into the shady realm of undefined behaviour.“

Ein vernünftiges Singleton sollte das *dead reference problem* wenigstens erkennen:

```
// Singleton.h
class Singleton
{
public:
   static Singleton& Instance() {
      if (!pInstance_)
      {
         if (destroyed_)
         {
            OnDeadReference();
         }
         else
         {
            Create();
         }
      }
      return *pInstance_;
   }
```

14. Singletons

```
// und weiter
private:
  static void Create() {
      static Singleton theInstance;
      pInstance_ = &theInstance;
   }
   static void OnDeadReference()
   {
      throw std::runtime_error("dead reference problem");
   }
   ~Singleton() {
      pInstance_ = 0;
      destroyed_ = true;
   }
   static Singleton* pInstance_;
   static bool destroyed_;
   ... deaktivierte 'tors/operator= ...
};
// Singleton.cpp
Singleton* Singleton::pInstance_ = 0;
bool Singleton::destroyed_ = false;
```

14. Singletons

Ist für das KDL Szenarion leider auch nicht zufriedenstellend ☹

Schritt 4 ezeugt die Exception:

„We got rid of the undefined behaviour; now we have to deal with unsatisfactory behaviour“

Log sollte IMMER verfügbar sein, ggf. sollte es (wie Phönix aus der Asche) wieder auferstehen ...

The Phoenix Singleton:

```
class Singleton
{
   ... wie zuvor...
   static void KillPhoenixSingleton();
};
void Singleton::OnDeadReference() {
   /*Re-*/Create();
   new(pInstance) Singleton;
   atexit(KillPhoenixSingleton); // for the next cleanup
   destroyed_ = false;
}
```

14. Singletons

The Phoenix Singleton:

```
// class Singleton cont.
  void Singleton::KillPhoenixSingleton() {
  // Asche zu Asche
  pInstance_->~Singleton();
}
```

Eigentlich ist damit nur eine spezielle Lösung für einen Sonderfall gefunden: `Log` soll alle anderen überleben.

„We need an easy way to control the lifetime of various singletons…
But wait, there‘s more: This problem applies not only to singletons but also to global objects in general. „

Wir brauchen *longevity control.*

14. Singletons

longevity control

```
// Eine Singleton-Klasse
class SomeSingleton { ... };
// Eine reguläre Klasse
class SomeClass { ... };
SomeClass* pGlobalObject(new SomeClass);
int main() {
   SetLongevity(&SomeSingleton().Instance(), 5);
   // pGlobalObject wird nach der Instanz
   // von SomeSingleton zerstört.
   SetLongevity(pGlobalObject, 6);
   ...
}

template <typename T>
void SetLongevity(T* pDynObject, unsigned int longevity);
```

14. Singletons

longevity control

NICHT für Objekte, deren Lebensdauer von Compiler bestimmt wird (global, static, automatic) – nur mittels `new` erzeugte Objekte

Nach `SetLongevity` darf man selbst auch nicht mehr `delete` rufen!

Die Lösung mit festen Zahlen hat einen entscheidenden Vorteil ggü. der zunächst eleganter aussehenden:

```
class DependencyManager
{
public:
   template <typename T, typename U>
   void SetDependency(T* dependent, U& target);
   ...
};
```

14. Singletons

Beide Objekte müssen bereits existieren (siehe `Log` ☹)

Idee: Die Abhängkeit erst in `Log` - Konstruktor herstellen?

„This, however, tightens the coupling between `Keyboard` and `Log` to an unacceptable degree … (*cyclic dependency*)“

Implementierung von `SetLongevity`:

1. **Jeder Aufruf plant den Destruktor per atexit.**
2. **Destruktion in der Reihenfolge aufsteigender *longevity.***
3. **Bei gleicher *longevity* wie in C++ üblich: LIFO**

→ schreit nach einer versteckten priority queue.

14. Singletons

```
namespace Private {
   class LifetimeTracker {
   public:
      LifetimeTracker(unsigned int x) : longevity_(x) {}
      virtual ~LifetimeTracker() = 0;
      friend inline bool Compare(
         unsigned int longevity,
         const LifetimeTracker* p)
      { return p->longevity_ < longevity; }
   private:
      unsigned int longevity_;
   };
   // Definition erforderlich
   inline LifetimeTracker::~LifetimeTracker() {}
   typedef LifetimeTracker** TrackerArray;
   extern TrackerArray pTrackerArray;
   extern unsigned int elements;
}
```

14. Singletons

`pTrackerArray` muss selbst Singleton sein.

Ein Huhn/Ei-Problem: **`SetLongevity`** muss zu jeder Zeit aufrufbar sein und sich damit gewissermaßen selbst verwalten – als letzen Mohikaner.

Lösung: *low level functions* aus der **`std::malloc`** - Familie:

```
template <typename T>
struct Deleter {
   static void Delete(T* pObj) { delete pObj; }
};
// Das konkrete Tracker-Objekt für Objekte vom Typ T
template <typename T, typename Destroyer>
class ConcreteLifetimeTracker : public LifetimeTracker {
public:
      ConcreteLifetimeTracker(T* p, unsigned int longevity, Destroyer d)
         :LifetimeTracker(longevity), pTracked_(p), destroyer_(d) {}
      ~ConcreteLifetimeTracker() {destroyer_(pTracked_);}
   private:
      T* pTracked_;
      Destroyer destroyer_;
   };
   void AtExitFn(); // s.u.
}
```

14. Singletons

```
template <typename T, typename Destroyer>
void SetLongevity(T* pDynObject, unsigned int longevity,
   Destroyer d = Private::Deleter<T>::Delete) {
   TrackerArray pNewArray = static_cast<TrackerArray>(
       std::realloc(pTrackerArray,
                    sizeof(LifeTimeTracker*)*(elements+1)));
   if (!pNewArray) throw std::bad_alloc();
   pTrackerArray = pNewArray;
   LifetimeTracker* p = new ConcreteLifetimeTracker<T, Destroyer>
                        (pDynObject, longevity, d);
   TrackerArray pos = std::upper_bound(
      pTrackerArray, pTrackerArray + elements, longevity, Compare);
   std::copy_backward(pos, pTrackerArray + elements,
      pTrackerArray + elements + 1);
   *pos = p;
   ++elements;
   std::atexit(AtExitFn);
}
```

14. Singletons

```
static void AtExitFn()
{
   assert(elements > 0 && pTrackerArray != 0);
   // Das oberste Element aus dem Stack auswählen
   LifetimeTracker* pTop = pTrackerArray[elements - 1];
   // Dieses Objekt aus dem Stack entfernen
   // Keine Fehlerüberprüfung - realloc mit weniger
   // Speicher kann nicht fehlschlagen
   pTrackerArray = static_cast<TrackerArray>
     (std::realloc(pTrackerArray, sizeof(LifetimeTracker*) * --elements));
   // Das Element zerstören
   delete pTop;
}
```

14. Singletons

Anwendung auf KDL-Szenario:

```
class Log
{
public:
   static void Create()
   {
      // Die Instanz erzeugen
      pInstance_ = new Log;
      // Hinzugefügt:
      SetLongevity(pInstance_, longevity_);
   }
   // Rest der Implementierung weggelassen
   // Log::Instance bleibt unverändert
private:
   // Definition eines festen Wertes für
   // die lange Lebensdauer.
   static const unsigned int longevity_ = 2;
   static Log* pInstance_;
};

// Keyboard/Display mit longevity 1
```

14. Singletons

In einer multi-threaded Welt ist unsere Singleton-Lösung leider defekt:

```
Singleton& Singleton::Instance()
{
   if (!pInstance_)                    // 1
   {
      pInstance_ = new Singleton;      // 2
   }
   return *pInstance_;                 // 3
}
```

Thread1 findet pInstance_ == 0 vor, tritt in den Block ein und wird in diesem Moment vom Scheduler angehalten.

Thread2 kommt dran, findet immer noch pInstance_ == 0 vor, erzeugt das Singleton ...

Irgendwann später setzt Thread1 fort mit //2, erzeugt das Singleton ... ☹

A race condition.

14. Singletons

A race condition.

Löst man üblicherweise mit einem Lock:

```
Singleton& Singleton::Instance()
{
   // mutex_ ist ein Mutex-Objekt
   // Lock verwaltet das Mutex-Objekt
   Lock guard(mutex_);
   if (!pInstance)
   {
      pInstance_ = new Singleton;
   }
   return *pInstance_;
}
```

Funktioniert, wird aber zum Flaschenhals: die Sperre wird nur einmal benötigt, sie ist aber auch in jedem späteren Aufruf (überflüssigerweise) dabei ☹

14. Singletons

Der Versuch einer Verbesserung:

```
Singleton& Singleton::Instance()
{
   if (!pInstance_)
   { // <- first thread stoopping here
      Lock guard(mutex_);
      pInstance_ = new Singleton;
   }
   return *pInstance_;
}
```

Bringt die *race condition* zurück ☹

"This seems like one of these brainteasers with no solution, but in fact there is a very simple and elegant one. It's called the Double-Checked Locking pattern (Doug Schmidt, Tim Harrison). "

14. Singletons

Double-Checked Locking pattern DCLP (Doug Schmidt, Tim Harrison)

```
Singleton& Singleton::Instance()
{
   if (!pInstance_)           // 1
   {                          // 2
      Lock myGuard(mutex_);   // 3 take the lock
      if (!pInstance_)        // 4 and test once more
      {
         pInstance_ = new Singleton;
      }
   }
   return *pInstance_;
}
```

Genial, aber leider immer noch nicht 100 % sicher ☹

Dr. Dobbs Journal, 2004

Scott Meyers, Andrei Alexandrescu: *C++ and the Perils of Double-Checked Locking* (Part I July and II August)

14. Singletons

C++ and the perils of Double-Checked Locking

```
pInstance_ = new Singleton;
```

Ist keine atomare Operation (wrt. threading):

Sie besteht aus drei Teiloperationen:

1. **Speicherplatz für das Singleton-Objekt anfordern**
2. **Den Konstruktor auf diesem Speicherplatz laufen lassen**
3. **Den Zeiger (`pInstance_`) auf diesen Speicherplatz richten.**

Compiler müssen diese Schritte nicht zwingend in dieser Reihenfolge ausführen. U.U. darf 2 NACH 3 ausgeführt werden:

```
Singleton& Singleton::Instance() {
   if (pInstance == 0) {
      Lock lock;
      if (pInstance ==_0) { // re-check
         pInstance_ = // Step 3
            operator new (sizeof(Singleton)); // Step1
  // ----- first thread interrupted here ------------
            new (pInstance_) Singleton; // Step 2
  }}   return *pInstance_;
}
```