C++ pre-/postAction

Technical Concept Paper

Reto Carrara
Gyrenstrasse 17
CH-8967 Widen
+4179/328’17’05
www.carrara.ch
info@carrara.ch

Inhalt

Inhalt 2
Revision 3
Einführung 4
Standard-Lösung....5
Pre-Action mit dem operator-> 7
Pre- und Post-Action....9
Pre- und Post-Action für beliebige Klassen....12
Pre- und Post-Action als Strategie....14
Strategie mit Status....17
Strategie für den synchronisierten Funktions-Aufruf 21
Schlussüberlegungen....22
Literatur....23

Revision

Datum	Version	Autor	Änderungen
14.02.02	1.0	carr	Basisversion

Einführung

Pre- und Post-Processing ist in seiner einfachsten Form ein Funktionsaufruf vor und nach einem Statement. Das vorliegende Paper diskutiert einen generischen Ansatz für Pre- und PostActions bei C++-Methoden.

Mit Hilfe von Templates und Template-Funktionen wird eine wiederverwendbare und intuitiv anwendbare Funktionalität entwickelt.

Pre- und Post-Actions lassen sich unter anderem zur Unterstützung von folgenden Komponenten einsetzen:

- Semaphore (Mutex-Section)
- Trace
- Debug-Output
- Assertions
- Zeitmessungen für Funktionsaufrufe

Dank einer sehr effizienten Implementation findet die gezeigte Lösung auch in Embedded-Systems Anwendung.

Es ist möglich, die Funktionalität ein- oder auszuschalten, wobei der Applikations-Code dazu nicht verändert werden muss. Auch die lästige Instrumentierung mit #ifdef Präprozessor-Anweisungen im Applikations-Code entfällt.

Standard-Lösung

Pre- und Post-Processing ist in seiner einfachsten Form ein Funktionsaufruf vor und nach einem Statement.

```
User aUser("User Object");

preAction();
aUser.fn1("function call");
postAction();

preAction();
aUser.fn2("function call");
postAction();
```

Die gezeigte Vorgehensweise führt aber unweigerlich zu Redundanz im Applikations-Code, muss doch der Aufruf für die Pre- und Post-Action immer wieder aufs neue erfolgen.

Die logische Schlussfolgerung ist nun, die Pre- und Post-Methoden in der benutzerdefinierten Methode (im Beispiel fn1 und fn2) zu platzieren.

Dies kann über eine Interface-Deklaration oder über Funktionszeiger erfolgen:

```
class User
{
    typedef void (*T_pFn)();
  public:
    User(const char* const i_pcString, T_pFn i_pFnPre, T_pFn i_pFnPost)
    : m_pcString (i_pcString)
    , m_pFnPre   (i_pFnPre)
    , m_pFnPost  (i_pFnPost)
    {}

    void fn1(const char* const i_pcString)
    {
      (*m_pFnPre)();
      cout << "1) " << m_pcString << ": " << i_pcString << endl;
      (*m_pFnPost)();
    }

    void fn2(const char* const i_pcString)
    {
      (*m_pFnPre)();
      cout << "2) " << m_pcString << ": " << i_pcString << endl;
      (*m_pFnPost)();
    }

  private:
    const char* const m_pcString;
    T_pFn m_pFnPre;
    T_pFn m_pFnPost;
};
```

Der Aufruf beschränkt sich nun auf die eigentliche Funktion:

```
User aUser("User Object",&preAction,&postAction);

aUser.fn1("function call");
aUser.fn2("function call");
```

Die redundanten Pre- und Post-Aufrufe wurden nun wohl aus dem Applikations-Code entfernt, trotzdem müssen sie für jede Methode (hier fn1 und fn2) wieder codiert werden.

Erstrebenswert wäre eine Lösung, welche für jeden Funktionsaufruf die Pre- und Post-Aufrufe automatisch generieren könnte.

Pre-Action mit dem operator->

Der Member-Access-Operator (->) kann bei C++ überladen werden und gibt einem Objekt Pointer-Charakteristik.

Das heisst, dass auf einem Objekt der Operator -> verwendet werden kann.

Beispiel:

```
class A
{
  public:
    void f() {...}
};

class B
{
  public:
    A* operator->() {...}
    void f() {...}
};

B b;

// Calls B::f
b.f();

// Calls A::f
b->f();
```

Der Member-Access-Operator bietet sich nun also an, um die Pre-Action zu platzieren:

```
class PreAction
{
  public:
    PreAction(User& i_raUser)
    : m_raUser (i_raUser)
    {}

    User* operator->()
    {
      preAction();
      return &m_raUser;
    }

  private:
    User& m_raUser;
    void preAction() {cout << "preAction" << endl;}
};
```

Die Pre-Action wird ausgeführt, bevor ein Zeiger auf das User-Objekt retourniert wird. Durch ein Überladen des operator-> wird auch erreicht, dass jede beliebige Methoden der Klasse User verwendet werden kann.

Der Aufruf gestaltet sich nun folgendermassen:

```
User aUser("User Object");

PreAction(aUser)->fn1("function call");
PreAction(aUser)->fn2("function call");
```

Durch die Anwendung des Member-Access-Operators wurde erreicht, dass jede Methode der Klasse User mit einer Pre-Action versehen werden kann. Dies ist von grossem Nutzen, da beliebig neue Methoden hinzugefügt werden können, ohne zusätzlichen Code zu schreiben.

Pre- und Post-Action

Da der Member-Access-Operator ein Pointer auf das User-Objekt zurückgibt ist es wohl einfach, eine Pre-Action-Methode zu implementieren, die Post-Action wird aber umso schwieriger.

```
User* operator->()
{
  preAction();
  return &m_raUser;
  ✍ postAction(); // Not possible
}
```

Im C++ Standard ISO/IEC 14882:1998(E) findet sich folgender Abschnitt zum Überladen des Member-Access-Operators:

13.5.6 Class member access
`operator->` shall be a non-static member function taking no parameters. It implements class member access using `->`

postfix-expression `->` *id-expression*

An expression `x->m` is interpreted as `(x.operator->())->m` for a class object `x` of type `T` if `T::operator->()` exists and if the operator is selected as the best match function by the overload resolution mechanism (13.3).

Wichtig dabei ist, dass `x->m` als `(x.operator->())->m` interpretiert wird. Das bedeutet, dass der Compiler als Rückgabewert des Member-Access-Operators ein Datentyp erwartet, auf welchem wiederum der Operator-> ausgeführt werden kann. Normalerweise ist dies natürlich ein Pointer. Es ist aber auch möglich, eine Referenz/Kopie eines Objektes zu retournieren, welches seinerseits ebenfalls den Operator-> überladen hat.

```
class A
{
  public:
    void f() {...}
};

class B
{
  public:
    A* operator->() {...}
};

class C
{
  public:
    B operator->() {...}
};

C c;
c->f();
```

Eine genauere Analyse des Statement `c->f()` ergibt folgendes:

```
c->f();
(c.operator->())->f();
```

wobei `c.operator->()` ein Objekt der Klasse B retourniert, dessen Operator-> aufgerufen wird.

```
((c.C::operator->()).B::operator->)->A::f();
```

So erstaunlich dies nun anmutet: geht aus dem kleinen unbedeutenden Abschnitt im Standard folgendes hervor:

> Wird im Member-Access-Operator ein Objekt retourniert, dessen Klasse seinerseits den Operator-> deklariert, so findet automatisch eine Kaskadierung der Aufrufe statt, ohne dass die Syntax des Initial-Aufrufes ändern würde.

Wie aus obigem Code ersichtlich ist, wird B „by value“ retourniert. Was bedeutet dies nun aber für den Lebenszyklus des retournierten Objektes?

Retourniert eine Funktion einen Wert, so ist dieser während des ganzen Funktionsaufrufes gültig. Daraus folgt, dass der Destruktor von B am Ende des Funktionsaufrufes ausgeführt werden muss. Und Ende bedeutet in diesem Kontext, dass die gesamte Befehlskette vollständig abgearbeitet wird.

Somit ist auch eine Lösung gefunden, wo die Pre- und Post-Action platziert werden soll; nämlich im Konstrutor, respektive im Destruktor von B.

Die Funktionalität des Systems kann also folgendermassen erweitert werden.

```
class Caller
{
  public:
    Caller(User& i_raUser)
    : m_raUser (i_raUser)
    {
      preAction();
    }

    ~Caller()
    {
      postAction();
    }

    User* operator->()
    {
      return &m_raUser;
    }

  private:
    User& m_raUser;

    void preAction() {cout << "preAction" << endl;}
    void postAction() {cout << "postAction" << endl;}
};

class PrePostAction
{
  public:
```

```
    PrePostAction(User& i_raUser)
    : m_raUser (i_raUser)
    {}

    Caller operator->()
    {
      return Caller(m_raUser);
    }

  private:
    User& m_raUser;

};
```

Der Aufruf erfolgt wie bereits gezeigt, obwohl nun sowohl die Pre- als auch die Post-Action ausgeführt werden.

```
User aUser("User Object");

PrePostAction(aUser)->fn1("function call");
PrePostAction(aUser)->fn2("function call");
```

An dieser Stelle noch ein wichtiger technischer Designhinweis. Da die Klasse Caller nur intern verwendet wird, kann sie auch als private Inner-Class deklariert werden. Somit wird bereits schon vom Compiler verhindert, dass die Klasse Caller für andere Zwecke missbraucht werden kann!

```
class PrePostAction
{
  private:
    class Caller
    {
      ....
    };
  public:
    PrePostAction(User& i_raUser)
    : m_raUser (i_raUser)
    {}

    Caller operator->()
    {
      return Caller(m_raUser);
    }

  private:
    User& m_raUser;

};
```

Pre- und Post-Action für beliebige Klassen

Was sich bis jetzt noch stark störend in Bezug auf die Wiederverwendbarkeit auswirkt ist der Umstand, dass die Pre-/Post-Action Funktionalität nur für die Klasse User Anwendung findet.

C++ bietet mit Hilfe von Templates aber eine einfache Möglichkeit, den Mechanismus für beliebige Klassen zu Verfügung zu stellen. Dazu muss lediglich die Klasse User durch einen generischen Typen ausgetauscht werden.

```
template <class T>
class PrePostAction
{
  private:
    class Caller
    {
      public:
        Caller(T& i_raObj)
        : m_raObj (i_raObj)
        {
          preAction();
        }

        ~Caller()
        {
          postAction();
        }

        T* operator->()
        {
          return &m_raObj;
        }

      private:
        T& m_raObj;

        void preAction() {cout << "preAction" << endl;}
        void postAction() {cout << "postAction" << endl;}
    };

  public:
    PrePostAction(T& i_raObj)
    : m_raObj (i_raObj)
    {}

    Caller operator->()
    {
      return Caller(m_raObj);
    }

  private:
    T& m_raObj;

};
```

Der Aufruf ändert sich nun natürlich dahingehend, dass der Template-Parameter spezifiziert werden muss.

```
User aUser("User Object");

PrePostAction<User>(aUser)->fn1("function call");
PrePostAction<User>(aUser)->fn2("function call");
```

Mit der Modifikation der Klasse PrePostAction steht die Mechanik für alle Klassen zur Verfügung.

Problematisch bleibt aber, dass die Methoden preAction und PostAction direkt in der Mechanik codiert sind. Dies darf aber nicht sein, soll die gezeigte Funktionalität als wiederverwendbare Komponente eingesetzt werden können.

Pre- und Post-Action als Strategie

Der Grundsatz des objekt-orientierten Design drückt sich sehr schön in der folgenden Definition aus:

> OOD ist die adäquate Trennung von Daten und Mechanik.

Diesem Grundsatz folgend wird der Code für das Pre- und Post-Processing ausgelagert. Trotzdem soll ein Aufruf so schnell wie möglich erfolgen können; das heisst, dass die Methoden nicht über ein Interface / eine Basisklasse abstrahiert werden sollen.

Da PrePostAction bereits ein C++-Template ist, bietet es sich an, die Strategie als Template-Parameter festzulegen.

```
template <class T, class S>
class PrePostAction
{
  private:
    class Caller
    {
      public:
        Caller(T& i_raObj)
        : m_raObj (i_raObj)
        {
          m_aStrategy.preAction();
        }

        ~Caller()
        {
          m_aStrategy.postAction();
        }

        T* operator->()
        {
          return &m_raObj;
        }

      private:
        T& m_raObj;
        S m_aStrategy;

    };

  public:
    PrePostAction(T& i_raObj)
    : m_raObj (i_raObj)
    {}

    Caller operator->()
    {
      return Caller(m_raObj);
    }

  private:
```

```
    T& m_raObj;

};
```

Wird der Caller für die Dauer des Funktionsaufrufes instanziert, so wird auch ein Strategie-Objekt vom Typ S angelegt. Dies bedeutet, dass im Objekt S ein Status angelegt werden kann, der in den Methoden pre- und postAction sowohl gelesen als auch geschrieben werden kann.

```
class PrePostStrategy
{
  public:
    void preAction() {cout << "preAction" << endl;}
    void postAction() {cout << "postAction" << endl;}
};
```

Für den Einsatz der erweiterten Funktionalität muss nun auch die gewünschte Strategie als Template-Parameter spezifiziert werden.

```
User aUser("User Object");

PrePostAction<User,PrePostStrategy>(aUser)->fn1("function call");
PrePostAction<User,PrePostStrategy>(aUser)->fn2("function call");
```

Eine Unschönheit sticht auch bei obiger Anwendung ins Auge! Obwohl der Typ der User-Klasse eigentlich durch die Parameter-Übergabe im Konstruktor determiniert ist, muss er bei der Template-Instazierung deklariert werden.

```
PrePostAction<User,PrePostStrategy>(aUser)->fn1("function call");
```

Es wäre ein Vorteil, wenn man die Parametrisierung des Templates auf das wesentliche beschränken könnte, das heisst nur die Strategie festlegen müsste:

```
User aUser("User Object");

Action<PrePostStrategy>(aUser)->fn1("function call");
Action<PrePostStrategy>(aUser)->fn2("function call");
```

Dies kann durch den geeigneten Einsatz von Template-Funktionen erreicht werden.

```
template <class S, class T>
PrePostAction<T,S> Action(T& i_raObj)
{
  return PrePostAction<T,S>(i_raObj);
}
```

Bei der Spezifikation der Template-Parameter wurde absichtlich die Reihenfolge der Parameter S und T vertauscht. Dies aus gutem Grund:

Template-Funktionen erlauben die automatische Deduktion[1] von Template-Parametern, wenn deren Typ eindeutig aus der Signatur der Funktionsparameter hervorgeht.

Obige Bedingung ist für den Typ T erfüllt, da dieser den Übergabe-Parameter i_raObj bestimmt. Typ S hingegen kann nicht automatisch deduziert werden, da er nicht als Parameter erscheint. Interessant an Template-Funktionen ist nun, dass nur diejenigen Template-Parameter spezifiziert werden müssen, welche vom Compiler nicht automatisch deduziert werden können. Diese müssen in der Liste der Template-Parameter an erster Stelle erscheinen.

Somit wird es also möglich, folgenden Aufruf zu tätigen:

```
Action<PrePostStrategy>(aUser)->fn1("function call");
```

Dabei handelt es bei Action um eine Template-Funktion, welcher die Strategie als Template-Parameter geliefert wird. Die notwendige Typeninformation für das Objekt, auf welchem eine beliebige Methode mit Pre- und Post-Actions versehen werden soll, wird dabei automatisch von der Template-Funktion deduziert und zum Generieren des entsprechenden PrePostAction-Objektes verwendet. Auf diesem kann, wie gewohnt, der Operator-> aufgerufen werden.

Der entwickelte Code erlaubt eine wiederverwendbare Mechanik, welche in jeder Basis-Bibliothek Platz findet. Der Aufruf ist durch die Anwendung von Templates und Template-Funktionen bequem und intuitiv.

Unterzieht man die Funktionsweise des Strategie-Objektes aber einer näheren Betrachtung, so muss man feststellen, dass der Status nur mit dem Default-Konstruktor initialisiert werden kann. Dies ist nur dann brauchbar, wenn das Strategie Singleton-Charakter hat, das heisst, keine weiteren Referenzen braucht, oder diese in einer statischen Variable ablegen kann.

Im folgenden soll aufgezeigt werden, wie das Strategie-Objekt je nach Situation initialisiert werden kann, aber nicht initialisiert werden muss. Um diese Funktionalität zu erreichen, müssen weitergehende Konzepte von C++-Templates eingesetzt werden.

[1] Deduktion: die vom Compiler automatisch bestimmte Typ-Information anhand der Übergabe-Parameter. ✍ Als Phil.: *Deduktion* oder *deduktiven Schluss* bezeichnet man den Schluss, der bei Wahrheit der Prämissen und Beachtung der Regeln der Logik die Wahrheit des Schlusssatzes gewährleistet.

Strategie mit Status

Könnte man dem Strategie-Objekt eine Referenz mitgeben, so wäre es zum Beispiel möglich, die Anzahl Aufrufe zu zählen, die Zeit für einen Aufruf zu messen oder andere Statistikaufgaben durchzuführen.

```
template <class T, class S>
class PrePostAction
{
  private:
    class Caller
    {
      public:
        Caller(T& i_raObj, S& i_raStrategy)
        : m_raObj      (i_raObj)
        , m_raStrategy (i_raStrategy)
        {
          m_raStrategy.preAction();
        }

        ~Caller()
        {
          m_raStrategy.postAction();
        }

        T* operator->()
        {
          return &m_raObj;
        }

      private:
        T& m_raObj;
        S& m_raStrategy;

    };

  public:
    PrePostAction(T& i_raObj);

    PrePostAction(T& i_raObj, const S& i_raStrategy);

    Caller operator->()
    {
      return Caller(m_raObj,m_aStrategy);
    }

  private:
    T& m_raObj;
    S m_aStrategy;

};
```

Um das Strategie-Objekt initialisieren zu können, wird der Copy-Constructor verwendet. Aus diesem Grund besitzt das PrePostAction-Template neu auch zwei Konstruktoren. Einer nimmt

wie bis anhin nur eine Referenz auf das Objekt, dessen Methoden mit der Pre-/Post-Action zu versehen sind.

Der zweite Konstruktor nimmt zusätzlich eine konstante Referenz auf ein Strategie-Objekt. Die Implementation der Konstruktoren ist einfach:

```
template <class T, class S>
PrePostAction<T,S>::PrePostAction(T& i_raObj)
: m_raObj (i_raObj)
{}

template <class T, class S>
PrePostAction<T,S>::PrePostAction(T& i_raObj, const S& i_raStrategy)
: m_raObj     (i_raObj)
, m_aStrategy (i_raStrategy)
{}
```

Dass die Konstruktoren explizit als outline implementiert sind hat einen guten Grund. Der erste Konstruktor verwendet nämlich einen Default-, der zweite hingegen einen Copy-Konstruktor. Nun ist es meist so, dass nicht immer ein Default-Konstruktor zur Verfügung gestellt werden kann und nicht immer ein Copy-Konstruktor geschrieben werden will.

> Werden Methoden von Templates outline implementiert, so werden diese nur kompiliert, wenn sie auch tatsächlich verwendet werden.

Somit muss im Strategie-Objekt wirklich nur ein Default-Konstruktor zur Verfügung gestellt werden, wenn der erste Konstruktor von PrePostAction aufgerufen wird. Wie bequem dieser Umstand ist, zeigt sich im täglichen Gebrauch.

Als Beispiel-Anwendung soll ein Zähler dienen, der zeigt, wie oft ein Funktionsaufruf stattgefunden hat.

Das Strategie-Objekt benötigt dazu eine Referenz auf einen integralen Wert. Natürlich darf auch eine korrekte Implementation des Copy-Konstruktors nicht fehlen, da dieser vom Konstruktor der Klasse PrePostAction vorausgesetzt wird.

```
class PrePostCountStrategy
{
  public:

    PrePostCountStrategy(int& i_raCallCounter)
    : m_raCallCounter (i_raCallCounter)
    {}

    PrePostCountStrategy(const PrePostCountStrategy& i_raStrategy)
    : m_raCallCounter (i_raStrategy.m_raCallCounter)
    {}

    void preAction()
    {
      ++m_raCallCounter;
      cout << "preAction" << endl;
    }
    void postAction()
```

```
    {
      cout << "postAction" << endl;
    }

  private:

    int& m_raCallCounter;
};
```

Nun folgt auch noch das Anpassen der Template-Funktion, welche für den bequemeren Aufruf zuständig ist.

```
template <class T, class S>
PrePostAction<T,S>::PrePostAction(T& i_raObj)
: m_raObj (i_raObj)
{}

template <class T, class S>
PrePostAction<T,S>::PrePostAction(T& i_raObj, const S& i_raStrategy)
: m_raObj     (i_raObj)
, m_aStrategy (i_raStrategy)
{}
```

Dem aufmerksamen Leser fällt wahrscheinlich sofort auf, dass nun auch der Template-Parameter S in der Parameter-Liste zu finden ist und sich somit automatisch durch den Compiler deduzieren lässt.

```
int nCounter = 0;
User aUser("User Object");

Action(aUser, PrePostCountStrategy(nCounter)}>fn1("function call");
Action(aUser, PrePostCountStrategy(nCounter)}>fn2("function call");
```

Dies mag im ersten Augenblick als angenehm erscheinen, bewirkt aber, dass Action auf zwei verschiedene Arten aufgerufen werden kann. Natürlich ist es weiterhin möglich, den Typ der Strategie zu spezifizieren:

```
Action<MyStrategy>(aUser, MyStrategy(....)}>fn(....);
```

Betrachtet man die Deklaration der Action-Funktion genauer, stellt man fest, dass genau die Strategie erwartet wird, welche auch als Template-Parameter übergeben wird:

```
template <class T, class S>
PrePostAction<T,S>::PrePostAction(T& i_raObj, const S& i_raStrategy)
```

Ruft man sich in Erinnerung, was die Spezialität eines Kontruktors mit genau einem Argument ist, so fällt auf, dass dieser zur impliziten Konversion verwendet werden kann.

Der Standard sagt dazu folgendes:

12.3.1 Conversion by constructor
A constructor declared without the *function-specifier* `explicit` that can be called with a single parameter specifies a conversion from the type of its first parameter to the type of its class. Such a constructor is called a converting constructor. *[Example:*

```
class X {
```

```
   // …
  public:
    X(int);
    X(const char*, int =0);
};
void f(X arg)
{
  X a = 1; // a = X(1)
  X b = "Jessie"; // b = X("Jessie",0)
  a = 2; // a = X(2)
  f(3); // f(X(3))
}
```
—*end example*]

Dieselbe Funktionalität lässt sich verwenden, um eine bequeme Initialisierung des Strategie-Status zu erreichen.

Da die PrePostCountStrategy über einen Konstruktor verfügt, der zur Umwandlung verwendet werden kann und die Template-Funktion als zweiter Parameter bereits ein Objekt vom Typ PrePostCountStrategy erwartet, wird eine implizite Konversion durchgeführt.

```
int nCounter = 0;
User aUser("User Object");

Action<PrePostCountStrategy>(aUser, nCounter}>fn1("function call");
Action<PrePostCountStrategy>(aUser, nCounter}>fn2("function call");
```

Auf die gezeigte Art und Weise gelingt es, den Status-Parameter für jede Strategie neu zu adaptieren, falls ein Converting-Constructor zur Verfügung steht.

Ebenfalls kann es für nicht versierte Template-Programmierer von Vorteil sein, dass der Aufruf von Action immer mit derselben Anzahl Template-Parameter erfolgt.

Strategie für den synchronisierten Funktions-Aufruf

Die Pre-/Post-Action Mechanik eignet sich vorzüglich, um vor dem Aufruf einer Funktion eine Semaphore anzufordern und nach dem Aufruf wieder freizugeben. Damit erreichen wir, dass Methoden auf einfache Art synchronisiert werden können.

Es bietet sich daher an, eine Semaphore-Strategie zur Verfügung zu stellen. Dies geschieht auf dieselbe Art und Weise wie der vorhin entwickelte Funktionsaufrufezähler, nur dass der Strategie jetzt eine Referenz auf die entsprechende Semaphore mitgegeben werden muss.

```
class Semaphore
{
  public:
    void aquire() {cout << "Semaphore::aquire()" << endl;}
    void release() {cout << "Semaphore::release()" << endl;}
};

class SemaphoreStrategy
{
  public:

    SemaphoreStrategy(Semaphore& i_raSemaphore)
    : m_raSemaphore (i_raSemaphore)
    {}

    SemaphoreStrategy(const SemaphoreStrategy& i_raStrategy)
    : m_raSemaphore (i_raStrategy.m_raSemaphore)
    {}

    void preAction()
    {
      m_raSemaphore.aquire();
    }
    void postAction()
    {
      m_raSemaphore.release();
    }

  private:
    Semaphore& m_raSemaphore;
};
```

Die Anwendung ist wiederum intuitiv und einfach:

```
Semaphore aSemaphore;

User aUser("User Object");

Action<SemaphoreStrategy>(aUser, aSemaphore)->fn1("function call");
Action<SemaphoreStrategy>(aUser, aSemaphore)->fn2("function call");
```

Schlussüberlegungen

Die gezeigte Technik erlaubt es, mit wenig Aufwand die Mechanik für Pre- und Post-Actions für C++-Methoden zur Verfügung zu stellen. Dank der Anwendung von Templates gelingt ein sehr generischer Ansatz.

Die Anwendung des Codes ist einfach und intuitiv. Aber auch die Entwicklung neuer Strategien kann von jedermann bewerkstelligt werden.

Da die Strategie als Template-Parameter deklariert ist, fallen langsame Virtual Calls weg, was gerade im Umfeld der Echtzeit-Programmierung wichtig ist. Werden die pre- und postAction-Methoden als inline deklariert, so entfällt sogar der Funktionsaufruf.

Eine leere Strategie kann somit von einem Compiler komplett wegoptimiert werden.

```
class VoidStrategy
{
  public:
    inline void preAction() {}
    inline void postAction() { }
};
```

Ärgerlich ist nur, dass das PrePostAction-Template trotzdem einen Caller instanziert, obwohl dies ja nicht nötig wäre, da VoidStrategy keinen Code enthält. Durch eine sogenannte partielle Template-Spezialisierung lässt sich aber auch hier eine Optimierung vornehmen.

Die momentan gewählte Strategie kann nun in einem Header-File festgelegt werden:

```
#ifdef _DEBUG
  typedef TraceStrategy T_TraceStrategy;
#else
  typedef VoidStrategy T_TraceStrategy;
#endif
```

Die Strategie wird dabei je nach Präprozessorschalter gewählt. Der grosse Vorteil an dieser Variante ist, dass nicht der gesamte Applikations-Code mit #ifdef instrumentiert werden muss.

```
Action<T_TraceStrategy>(aUser)->fn1("function call");
```

Literatur

Gamma, Erich et al.: Design Patterns. Addison Wesley, 1995. ISBN 0-201-63361-2.
Ein Standardwerk, das in keiner Bibliothek fehlen sollte. Vor allem als Nachschlagewerk und Lebensgefährte gedacht. Das beste Buch auf seinem Gebiet! Sehr empfehlenswert für jedermann.

Stroustrup, Bjarne: The C++ Programming Language, Third Edition. Addison Wesley, 1997. ISBN 0-201-88954-4.
Die beste C++ Referenz. Achtung: Nur die dritte Edition kaufen. Exzellente Einführung in STL. Sehr empfehlenswert für jedermann.

Lippmann, Stanley B., Lajoie, Josée: C++ Primer, Third Edition. Addison Wesley, 1998. ISBN 0-201-82470-1.
Ebenfalls ein sehr gutes C++ Buch. Einfacher zu verstehen als das Buch Stroustrup. Auch hier: Nur die dritte Edition kaufen. Gute STL Referenz

Lippmann, Stanley B: Inside the C++ Object Model. Addison Wesley, 1996. ISBN 0-201-83454-5.
Ein geniales und einfach geschriebenes Buch, für alle die wissen wollen, wie C++ intern funktioniert. Vom "virtual call" bis hin zu "multiple inheritance RTTI".

Coplien, James O.: Advanced C++ Programming Styles and Idioms. . Addison Wesley, 1992. ISBN 0-201-54855-0.
Ein Buch mit weiterführenden Konzepten. Zum Teil hochtechnische und sehr schwierig zu verstehende Lösungsansätze. Die Perle unter den C++ Büchern.

Meyers, Scott: Effective C++, Second Edition. Addison Wesley, 1998. ISBN 0-201-92488-9.
Meyers, Scott: More Effective C++. Addison Wesley, 1998. ISBN 0-201-63371-X.
Zwei Bücher, die sich mit dem korrekten Schreiben von C++ Programmen befassen. So zum Beispiel, wie man Assignment Operators, Copy Constructors und deren mehr korrekt implementiert. Sehr empfehlenswert für jedermann.

Stroustrup, Bjarne: The Annotated C++ Reference Manual. Addison Wesley, 1995. ISBN 0-201-51459-1.
Die C++ Referenz für Compilerbauer. Geht sehr tief und ist mit vielen Bemerkungen versehen.