Exceptional C++

Professionelle Programmierung

PEARSON Education

ADDISON-WESLEY

Herb Sutter

Exceptional C++

47 technische Denkaufgaben, Programmierprobleme und ihre Lösungen

Deutsche Übersetzung von Mathias Born und Michael Tamm

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
der Deutschen Bibliothek erhältlich.

Der Titel der amerikanischen Originalausgabe lautet: *Herb Sutter, Exceptional C++: 47 engineering puzzles, programming problems, and solutions. Addison Wesley Longman, Inc., USA, 2000 ISBN 0–201–61562–2*

10 9 8 7 6 5 4 3 2 1

03 02 01 00

ISBN 3-8273-1711-8

Einbandgestaltung: viervierte Gestaltung, Köln
Übersetzung: Mathias Born und Michael Tamm, Berlin
Lektorat: Christina Gibbs, cgibbs@pearson.de
Korrektorat: Simone Burst, Großberghofen
Herstellung: TYPisch Müller, Arcevia, Italien
Satz: reemers publishing services gmbh, Krefeld
Druck und Verarbeitung: Schoder, Gersthofen
Printed in Germany

Gewidmet meiner Familie für ihre geduldige Unterstützung, ebenso Eric Wilson, Jeff Summer, Juana Chang, Larry Palmer, dem Rest der Entwicklerkerngruppe bei PeerDirect und meiner »vergrößerten« Familie der letzten vier Jahre:

Declan West
Doug Shelley
Duk Loi
Ian Long
Justin Karabegovic
Kim Nguyen
Margot Fulcher
Mark Cheers
Morgan Jones
Violetta Lukow

Dank an euch für alles.

Inhalt

Vorwort

Dies ist ein bemerkenswertes Buch. Allerdings habe ich erst, nachdem ich es fast zu Ende gelesen hatte, gemerkt, wie bemerkenswert es ist. Gut möglich, dass dies überhaupt das erste Buch ist, das jemals für Leute geschrieben wurde, die bereits mit C++ vertraut sind – mit *allen* Seiten von C++. Von Sprachmerkmalen über Komponenten der Standardbibliothek bis zu Programmiertechniken, dieses Buch springt von Thema zu Thema und bringt Sie immer ein klein wenig aus dem Gleichgewicht, damit Sie aufmerksam dranbleiben. Eben genau wie echte C++-Programme. Klassendesign trifft auf virtuelle Funktionen, Iteratorkonventionen treffen auf Namensraumregeln, Zuweisungsoperatoren streifen Exception-Sicherheit, Übersetzungsabhängigkeiten kreuzen sich mit exportierten Templates. Genau wie in echten Programmen. Das Ergebnis ist ein zugleich chaotischer und großartiger, Schwindel erregender Strudel aus Sprachmerkmalen, Bibliothekskomponenten und Programmiertechniken. Eben wie echte Programme.

Ich betone *GotW* so, dass es sich auf »Gotcha« reimt, und das passt ganz gut. Beim Vergleich meiner Lösungen auf die im Buch gestellten Fragen mit Sutters Lösungen stellte ich fest, dass ich öfter, als ich zugeben möchte, in die Fallen getappt bin, die er (und C++) aufgestellt hatte. Bei jedem Fehler, den ich beging, konnte ich beinahe Herb lächeln sehen und leise »Gotcha!« sagen hören. Einige würden daraufhin argumentieren, ich verstünde nicht viel von C++. Andere würden sagen, dies zeige, dass C++ zu komplex ist, um von irgendjemandem beherrscht zu werden. Ich persönlich denke, dass dies zeigt, dass man sehr genau darüber nachdenken muss, was man tut, wenn man mit C++ arbeitet. C++ ist eine mächtige Sprache, die zur Lösung anspruchsvoller Probleme geschaffen wurde. Deshalb ist es wichtig, dass Sie so gut wie Sie können an Ihrer Kenntnis der Sprache, ihrer Bibliothek und ihrer Programmieridiome feilen. Die Themenbandbreite und das einmalige Quizformat dieses Buches werden Ihnen dabei behilflich sein.

Stammleser der C++-Newsgruppen wissen, wie schwierig es ist, zum *Guru of the Week* ernannt zu werden. Und die alteingesessenen Teilnehmer wissen das umso besser. Im Internet kann es natürlich nur einen Guru pro Woche geben. Aber mit den Informationen aus diesem Buch können Sie darauf hoffen, in Zukunft bei Ihren Programmen Code von Guru-Qualität zu erzeugen.

Scott Meyers
Juni 1999

1 Einleitung

Exceptional C++ zeigt anhand von Beispielen, wie man solides Software-Engineering in Angriff nimmt. Zusammen mit einer Menge anderen Materials enthält dieses Buch erweiterte Versionen der ersten 30 Ausgaben der populären Internet-C++-Reihe *Guru of the Week* (kurz *GotW*), in der abgeschlossene C++-Probleme und deren Lösungen vorgestellt werden, um bestimmte Design- und Programmiertechniken zu demonstrieren.

Dieses Buch ist keine wahllose Ansammlung von Code-Puzzles, sondern in erster Linie ein Leitfaden für vernünftiges Unternehmens-Software-Design in C++, das in der realen Welt Bestand hat. Es setzt dabei auf ein Problem/Lösungs-Format, da dies die effektivste Methode ist, die ich kenne, um Sie, verehrter Leser, in die Gedanken hinter den Problemen und die Gründe für die Richtlinien einzubeziehen. Die Lektionen decken zwar viele Themen ab, aber Sie werden feststellen, dass sich viele davon auf Aspekte der Unternehmensentwicklung konzentrieren, insbesondere auf Exception-Sicherheit, vernünftiges Klassen- und Modul-Design, geeignete Optimierung und portablen, standardkonformen Code.

Ich hoffe, dass dieser Stoff Ihnen bei Ihrer täglichen Arbeit behilflich ist. Aber ich hoffe auch, dass Sie wenigstens ein paar pfiffige Inspirationen und elegante Techniken entdecken und dass Sie während des Lesens von Zeit zu Zeit ein plötzliches Aha-Erlebnis haben. Denn wer sagt schon, Software-Entwurf müsse langweilig sein?

Wie liest man dieses Buch

Ich erwarte, dass Sie mit den Grundlagen von C++ bereits vertraut sind. Sollte das nicht der Fall sein, beginnen Sie besser mit einer guten C++-Einführung (klassische Wälzer wie *The C++ Programming Language, Third Edition*[1] von Bjarne Stroustrup oder *C++ Primer, Third Edition*[2] von Josêe Lajoie sind eine gute Wahl) und befassen sich dann auf jeden Fall mit einem stilistischen Leitfaden wie Scott Meyers klassische Bücher *Effective C++* (Die Browser-basierte CD-Version fand ich recht bequem und nützlich).[3]

1. Stroustrup B.: *The C++ Programming Language, Third Edition* (Addison-Wesley Longman, 1997)
2. Lippman S. und Lajoie J.: *C++ Primer, Third Edition* (Addison-Wesley Longman, 1998)
3. Meyers S.: *Effective C++ CD: 85 Specific Ways to Improve Your Programms and Designs* (Addison-Wesley Longman, 1999). Eine Online-Demo ist unter http://www.meyerscd.awl.com verfügbar.

Jede Lektion in diesem Buch wird als Puzzle oder Problem präsentiert und hat einen einführenden Kopf, der so aussieht:

Lektion ##: Das Thema dieses Puzzles	Schwierigkeitsgrad: X

Das Thema und die Schwierigkeitsbewertung (gewöhnlich im Bereich 3 bis 9½ auf einer Skala von 1 bis 10) geben Ihnen einen Hinweis darauf, womit Sie es zu tun bekommen. Aber beachten Sie, dass der Schwierigkeitsgrad meine ganz persönliche Einschätzung drüber darstellt, wie schwer die meisten Leute ein Problem finden werden. Es kann daher sein, dass Ihnen ein 7er-Problem leichter fällt als ein anderes 5er-Problem. Trotzdem ist es besser, sich auf das Schlimmste vorzubereiten, wenn ein 9½-Monster ansteht.

Es ist nicht erforderlich, die Abschnitte in der Reihenfolge ihres Auftretens zu lesen, aber an einigen Stellen gibt es »Miniserien« verwandter Probleme, die sich durch ein »Teil 1«, »Teil 2« usw. – einige sogar bis »Teil 10« – im Namen auszeichnen. Diese Miniserien sollten am besten als Gruppe gelesen werden.

Wie kam es dazu: GotW und PeerDirect

Die C++-Serie *Guru of the Week* hat eine lange Geschichte. Urspünglich wurde *GotW* Ende 1996 ins Leben gerufen, um unserem eigenen Entwicklerteam hier bei PeerDirect interessante Herausforderungen und fortlaufende Ausbildung zu bieten. Ich wollte damit ein unterhaltsames Fortbildungswerkzeug zur Verfügung stellen, in dem auch Dinge wie die korrekte Benutzung von Vererbung und Exception-Sicherheit ausführlich zur Sprache kommen konnten. Mit der Zeit wurde es auch als Mittel benutzt, um unser Team mit den neuesten Ergebnissen der C++-Standardisierungskonferenzen vertraut zu machen. Als reguläres Merkmal der Internet-Newsgruppe *comp.lang.c++.moderated* wurde *GotW* dann schließlich der Öffentlichkeit zugänglich gemacht. Dort können Sie jede neue Problemstellung und entsprechende Antworten (sowie viele interessante Diskussionen) finden.

Aus den gleichen Gründen wie in Ihrer Firma ist die korrekte Benutzung von C++ auch bei PeerDirect sehr wichtig, auch wenn vielleicht verschiedene Ziele damit erreicht werden sollen. Wir bauen System-Software – für verteilte Datenbanken und Datenbankenreplikation –, in der unternehmerische Fragestellungen wie Zuverlässigkeit, Sicherheit, Portabilität, Effizienz und viele andere überlebenswichtig sind. Unsere Software muss auf verschiedene Compiler und Betriebssysteme portierbar sein; sie muss auch bei Deadlocks während Datenbanktransaktionen, bei Kommunikationsunterbrechungen und bei Programm-Exceptions sicher und robust arbeiten. Kunden ver-

walten damit alles von kleinen Datenbanken in Smart-Cards, auf PalmOS- oder WinCE-Geräten über Datenbanken auf Abteilungs-Servern unter Windows NT, Linux und Solaris bis hin zu massiv parallelen Oracle-Backends für Web-Server und Data-Warehouses – und all das mit der gleichen Software, der gleichen Zuverlässigkeit und dem gleichen Code. Nun, da wir uns einer halben Million knapper, unkommentierter Zeilen Code nähern, ist das eine Portabilitäts- und Zuverlässigkeitsherausforderung.

Denjenigen von Ihnen, die in den letzten Jahren den *Guru of the Week* im Internet verfolgt haben, möchte ich noch paar Dinge sagen:

- Vielen Dank für Ihr Interesse, Ihre Unterstützung, die E-Mails, den Ruhm, die Korrekturen, die Kommentare, die Kritik, die Fragen – und insbesondere für Ihre Anfragen nach einer *GotW*-Serie in Buchform. Hier ist sie nun, ich hoffe sie gefällt Ihnen.
- Dieses Buch enthält *weit mehr*, als Sie jemals im Internet gesehen haben.

Exceptional C++ wurde nicht einfach per Cut-and-Paste aus alten *GotW*-Folgen zusammengeschustert, die bereits irgendwo da draußen im Cyberspace kursieren. Alle Probleme und Lösungen wurden erneut durchgesehen und beträchtlich überarbeitet – zum Beispiel erschienen die Lektionen 8 bis 17 über Exception-Sicherheit ursprünglich als einzelnes *GotW*-Puzzle und sind nun zu einer tief gehenden, zehnteiligen Miniserie angewachsen. Jedes Problem und jede Lösung wurden wenn nötig aktualisiert, um dem heutigen, offiziellen C++-Standard zu entsprechen.

Es gibt hier also jede Menge Neues für Sie, auch wenn Sie die *GotW*-Serie zuvor regelmäßig verfolgt haben. Noch einmal vielen Dank an alle treuen Leser, und ich hoffe, dass dieses Material Ihnen dabei behilflich ist, Ihre Fähigkeiten in Software-Engineering und C++-Programmierung weiterhin zu vervollständigen und zu verfeinern.

Anerkennungen

Danke natürlich zuerst an all die *GotW*-Leser und -Enthusiasten in *comp.lang.c++.moderated*, insbesondere an diejenigen, die an dem Wettbewerb um den Namen für dieses Buch teilgenommen haben. Zwei Menschen trugen besonders dazu bei, uns zum endgültigen Titel zu führen, und ihnen möchte ich gesondert danken: Marco Dalla Gasperina für den Vorschlag *Enlightened C++* und Rob Stewart für seinen Vorschlag *Practical C++ Problems and Solutions*. Von da an war es nur noch ein Schritt, das Wortspiel *exceptional* hinzuzufügen, um damit den wiederholt auftretenden Schwerpunkt Exception-Sicherheit zu betonen. Vielen Dank auch an den Redakteur der Serie, Bjarne Stroustrup, und an Marina Lang, Debbie Lafferty und den Rest der Redaktion bei Addison-Wesley Longman für ihr kontinuierliches Interesse und ihren Enthusiasmus für dieses Projekt, und für den wirklich guten Empfang, den sie 1998 anlässlich der C++-Standardisierungskonferenz in Santa Cruz gegeben haben.

Ich möchte auch den vielen Menschen danken, die probegelesen haben – viele von ihnen sind wie ich Mitglieder des Standardisierungskommitees – und die wertvolle und präzise Hinweise gaben, durch die der Text, den Sie gleich lesen werden, verbessert werden konnte. Besonderen Dank an Bjarne Stroustrup, Scott Meyers, Andrei Alexandrescu, Steve Clamage, Steve Dewhurst, Cay Horstmann, Jim Hyslop, Brendan Kehoe und Dennis Mancl für ihre unbezahlbaren Einsichten und Kritiken.

Am meisten möchte ich schließlich meiner Familie und meinen Freunden danken, die auf so unterschiedliche Weise immer für mich da sind.

Herb Sutter
Juni 1999

2 Generische Programmierung und die Standard-C++-Bibliothek

Lassen Sie uns zu Beginn einige Aspekte generischer Programmierung betrachten. Dabei konzentrieren wir uns darauf, wie man Templates, Iteratoren und Algorithmen effektiv gebraucht sowie die Möglichkeiten der Standardbibliothek benutzt und erweitert. Das Ganze mündet dann schließlich im darauffolgenden Abschnitt, der sich mit Exception-Sicherheit speziell im Zusammenhang mit Templates befasst.

Lektion 1: Iteratoren **Schwierigkeitsgrad: 7**

Bei der Benutzung der Standardbibliothek muss sich jeder Programmierer der diversen häufig begangenen sowie auch der weniger häufig begangenen Fehler bewusst sein. Wie viele davon können Sie finden?

Das folgende Programm ist im Hinblick auf Iteratoren an mindestens vier Stellen problematisch. Wie viele sehen Sie?

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream _iterator<Date>(),
        back_in serter( e ) );
  vector<Date>::iterator first =
        find( e .begin(), e.end(), "01/01/95" );
  vector<Date>::iterator last =
        find( e .begin(), e.end(), "12/31/95" );
  *last = "12/30/95";
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
  e.insert( --e.end(), TodaysDate() );
  copy( first,
        last,
        ostream_iterator<Date>( cout, "\n" ) );
}
```

Lösung

```
int main()
{
  vector<Date> e;
  copy( istream_iterator<Date>( cin ),
        istream_iterator<Date>(),
        back_inserter( e ) );
```

Das ist soweit in Ordnung. Der Autor der `Date`-Klasse hat eine Extraktionsfunktion namens `operator>>( istream&, Date& )` bereitgestellt, die von `istream_iterator<Date>` benutzt wird, um die `Dates`-Objekte aus dem `cin`-Stream zu lesen. `copy()` überträgt diese dann lediglich in den `vector`.

```
vector<Date>::iterator first =
      find( e.begin(), e.end(), "01/01/95" );
vector<Date>::iterator last =
      find( e.begin(), e.end(), "12/31/95" );
 *last = "12/30/95";
```

Fehler: Dies könnte unzulässig sein, denn es besteht die Möglichkeit, dass `last` den Wert `e.end()` hat und somit nicht dereferenzierbar ist.

Der Algorithmus `find()` gibt sein zweites Argument zurück (also den Iterator, der im übergebenen Bereich das Ende kennzeichnet), wenn die Suche erfolglos war. `last` erhält also in diesem Fall, wenn »12/31/95« nicht im Container enthalten ist, den Wert `e.end()` und wird damit zu einem ungültigen Iterator, der auf die erste Stelle nach dem Ende des Container verweist.

```
copy(  first,
       last,
       ostream_iterator<Date>( cout, "\n" ) );
```

Fehler: Dies könnte unzulässig sein, da `[first, last)` eventuell keinen gültigen Bereich darstellt, denn es kann sein, dass `first` auf eine Stelle hinter `last` verweist.

Wenn zum Beispiel »12/31/95« in `e` gefunden wird, »01/01/95« dagegen aber nicht, dann zeigt der Iterator `last` irgendwo in den Container (eben auf das `Date`-Objekt, das gleich »12/31/95« ist), `first` jedoch auf `e.end()`. Nun setzt `copy()` aber voraus, dass `first` auf eine Position im Container verweist, die vor der von `last` referenzierten Stelle liegt. Der Bereich `[first, last)` muss daher zwingend ein gültiger Bereich sein.

Wenn Sie keine Fehler prüfende Version der Standardbibliothek verwenden, die einige dieser Probleme für Sie entdecken kann, äußern sich diese Fehler durch einen Programmabsturz während oder kurz nach `copy()`.

```
e.insert(  --e.end() , TodaysDate() );
```

Erster Fehler: Der Ausdruck »`--end()`« ist wahrscheinlich unzulässig.

Dafür gibt es einen einfachen Grund: Viele populäre Implementierungen der Standardbibliothek realisieren `vector<Data>::iterator` häufig schlicht als `Date*`. C++ erlaubt allerdings bei eingebauten Typen keine Veränderung von temporären Variablen. Zum Beispiel ist deshalb der folgende einfache Code falsch:

```
Date* f();  // Funktion, die ein Date* zurückgibt
p = --f();  // unzulässig, könnte aber "f()-1" sein
```

Nun wissen wir allerdings, dass `vector<Date>::iterator` ein Iterator mit wahlfreiem Zugriff (Random-Access-Iterator) ist. Wir können den Code daher ohne Effizienzverlust wie folgt verbessern:

```
e.insert( e.end() - 1, TodaysDate() );
```

Zweiter Fehler: Es gibt da noch ein anderes Problem. Wenn `e` leer ist, wird der Versuch, einen Iterator zu erzeugen, der vor `e.end()` zeigt (egal, ob nun vermittelst »`-e.end()`« oder »`e.end()-1`«), zu einem ungültigen Iterator führen.

```
copy( first,,
      last,,
      ostream_iterator<Date>( cout, "\n" ) );
}
```

Fehler: `first` und `last` könnten durchaus beides ungültige Iteratoren sein.

Ein `vector` wächst blockweise, damit sein Speicher nicht für jedes Einfügen realloziert werden muss. Irgendwann ist der `vector` aber voll, und das nächste Einfügen eines Elements erzwingt eine solche Reallozierung.

Als Folge des Aufrufs von `e.insert()` kann der `vector` also wachsen oder auch nicht, d.h. sein Speicher wird entweder bewegt oder nicht. Auf Grund dieser Unsicherheit müssen wir daher nach einer solchen Operation jeden existierenden Iterator dieses Containers als ungültig betrachten. Wenn in unserem Fall der Speicher tatsächlich bewegt werden würde, machte sich der fehlerhafte `copy`-Aufruf wieder als schwer zu erklärender Programmabsturz bemerkbar.

☑ Richtlinie

Dereferenzieren Sie niemals einen ungültigen Iterator.

Zusammenfassung: Beachten Sie bei der Verwendung von Iteratoren die folgenden vier Punkte:

1. Gültige Werte: Ist der Iterator dereferenzierbar? »`*e.end()`« stellt zum Beispiel immer einen Programmierfehler dar.

2. Lebensdauer: Ist der Iterator zum Zeitpunkt seiner Benutzung noch gültig? Oder wurde er durch eine Operation ungültig gemacht, die nach seiner Erzeugung durchgeführt wurde?

3. Gültige Wertebereiche: Wird durch ein Iterator-Paar ein gültiger Bereich gebildet? Ist `first` wirklich kleiner oder gleich `last`? Zeigen beide tatsächlich in den gleichen Container?

4. Unzulässige Veränderung von Instanzen eingebauter Datentypen: Versucht der Code, wie in dem obigen »--`e.end()`«-Beispiel, eine temporäre Variable zu manipulieren, die Instanz eines eingebauten Datentyps ist? (Zum Glück kann der Compiler diese Art Fehler häufig entdecken. Und bei Iteratoren, die als Klassen realisiert sind, erlaubt der Bibliotheksautor diese Vorgehensweise meistens im Interesse der syntaktischen Bequemlichkeit.)

Lektion 2: Von der Groß-/Kleinschreibung unabhängige Strings – Teil 1 — Schwierigkeitsgrad: 7

Sie wollen also eine String-Klasse, die die Groß-/Kleinschreibung ignoriert? Dann lautet Ihr Auftrag, falls Sie ihn übernehmen, eine zu schreiben.

In dieser Lektion gibt es drei relevante Aspekte.

1. Was bedeutet »von der Groß-/Kleinschreibung unabhängig«?

2. Schreiben Sie eine Klasse `ci_string`[1], die identisch zur Standardklasse `std::string` ist, jedoch in der gleichen Weise nicht von der Groß-/Kleinschreibung abhängt wie die auf vielen Systemen zu findende Erweiterung `stricmp()`.[2] Eine solche Klasse sollte sich wie folgt benutzen lassen:

```
ci_string s( "AbCdE" );
// Groß-/Kleinschreibung egal
//
assert( s == "abcde" );
assert( s == "ABCDE" );
// Groß-/Kleinschreibung bleibt aber
// natürlich erhalten
assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
assert( strcmp( s.c_str(), "abcde" ) != 0 );
```

3. Ist es günstig, die Unabhängigkeit von der Groß-/Kleinschreibung zu einer Objekteigenschaft zu machen?

1. Anm. d. Übers.: `ci` steht für case-insensitive.
2. Die von der Groß-/Kleinschreibung unabhängige String-Vergleichsfunktion `stricmp()` gehört nicht zum C- oder C++-Standard, ist aber eine bei vielen C- und C++-Compilern vorkommende Erweiterung.

Lösung

Die Antworten auf die drei Fragen lauten wie folgt.

1. Was bedeutet »von der Groß-/Kleinschreibung unabhängig«?

   Was das tatsächlich heißt, hängt ganz von der Anwendung und der Sprache ab. Viele Sprachen ignorieren zum Beispiel die Groß-/Kleinschreibung komplett. Bei denen, die sie beachten, müssen Sie immer noch entscheiden, ob Buchstaben mit Akzenten gleich denen ohne Akzenten sein sollen usw. Diese Lektion zeigt Ihnen, wie Sie die Nichtbeachtung der Groß-/Kleinschreibung für Standard-Strings implementieren können, unabhängig davon, welcher Fall nun konkret für Ihre Situation zutrifft.

2. Schreiben Sie eine Klasse `ci_string`, die identisch zur Standardklasse `std::string` ist, jedoch in der gleichen Weise nicht von der Groß-/Kleinschreibung abhängt wie die auf vielen Systemen zu findende Erweiterung `stricmp()`.

   Die Frage »Wie baue ich einen von der Groß-/Kleinschreibung unabhängigen String?« taucht so häufig auf, dass sie wahrscheinlich ihre eigene FAQ verdient – daher diese Lektion.

   Das wollen wir erreichen:

   ```
   ci_string s( "AbCdE" );
   // Groß-/Kleinschreibung egal
   //
   assert( s == "abcde" );
   assert( s == "ABCDE" );
   // Groß-/Kleinschreibung bleibt aber
   // natürlich erhalten
   assert( strcmp( s.c_str(), "AbCdE" ) == 0 );
   assert( strcmp( s.c_str(), "abcde" ) != 0 );
   ```

   Hier ist es nun wichtig zu verstehen, wie ein `string` im Standard-C++ aufgebaut ist. Wenn Sie einen Blick in Ihre `string`-Headerdatei werfen, dann sehen Sie darin etwas wie dies:

   ```
   typedef basic_string<char> string;
   ```

   `string` ist also tatsächlich keine Klasse, sondern ein mit `typedef` abgekürzter Template-Name. Das Template `basic_string<char>` selbst ist wiederum wie folgt definiert, eventuell mit weiteren implementierungsspezifischen Parametern:

   ```
   template<class charT,
            class traits = char_traits<charT>,
            class Allocator = allocator<charT> >
   class basic_string;
   ```

»string« bedeutet also in Wirklichkeit »basic_string<char, char_traits<char>, allocator<char> >«, optional mit zusätzlichen systemabhängigen Default-Template-Parametern. Das allocator-Argument braucht uns hier nicht weiter zu kümmern. Entscheidend ist der char_traits-Teil, da er festlegt, in welcher Beziehung die Zeichen zueinander stehen – und wie sie zu vergleichen sind!

Lassen Sie uns also strings vergleichen. Die Klasse basic_string stellt nützliche Vergleichsfunktionen zur Verfügung, mit denen man feststellen kann, ob ein string gleich einem anderen, kleiner als ein anderer usw. ist. Diese Vergleichsfunktionen benutzen Funktionen aus dem char_traits-Template, die wiederum einzelne Zeichen vergleichen. Im einzelnen sind dies die Elementfunktionen eq() und lt() zum Vergleich auf Gleichheit und auf »kleiner als« sowie compare() und find() zum Vergleich und zum Suchen ganzer Sequenzen von Zeichen.

Um das Verhalten dieser Funktionen zu verändern, müssen wir lediglich ein neues char_traits-Template bereitstellen. Die einfachste Vorgehensweise ist dabei diese:

```
struct ci_char_traits : public char_traits<char>
              // alle anderen Funktionen erben,
              // die wir nicht ersetzen müssen
{
  static bool eq( char c1, char c2 )
    { return toupper(c1) == toupper(c2); }
  static bool lt( char c1, char c2 )
    { return toupper(c1) <  toupper(c2); }
  static int compare( const char* s1,
                      const char* s2,
                      size_t n )
    { return memicmp( s1, s2, n ); }
          // sofern auf Ihrer Plattform verfügbar,
          // sonst einfach selber machen
  static const char*
  find( const char* s, int n, char a )
  {
    while( n-- > 0 && toupper(*s) != toupper(a) )
    {
        ++s;
    }
    return n > 0 ? s : 0;
  }
};
```

Alles, was wir damit getan haben, ist, mit typedef einen neuen String ci_string einzuführen, der genauso funktioniert wie der Standardstring string (in fast jeder Hinsicht *ist* es im Grunde ein string). Nur benutzt unser neuer String eben nicht char_traits<char>, sondern ci_char_traits, um die einzelnen Zeichen zu vergleichen. Weil ci_char_traits die Groß-/Kleinschreibung ignoriert, verhält sich

`ci_string` folgerichtig ebenso. Wir haben also ohne Eingriffe in `basic_string` einen String-Typ geschaffen, der die Groß-/Kleinschreibung nicht beachtet – *das* ist wahre Erweiterbarkeit.

3. Ist es günstig, die Unabhängigkeit von der Groß-/Kleinschreibung zu einer Objekteigenschaft zu machen?

   Meistens ist es nützlicher, die Unabhängigkeit von der Groß-/Kleinschreibung zu einer Eigenschaft der Vergleichsfunktion zu machen, anstatt, wie es hier gezeigt wurde, zu einer Eigenschaft des Objekts. Betrachten Sie zum Beispiel den folgenden Code:

   ```
   string    a = "aaa";
   ci_string b = "aAa";
   if( a == b ) /* ... */
   ```

   Welches Ergebnis soll nun, eine passende Funktion `operator==()` vorausgesetzt, der Ausdruck »`a == b`« liefern? Wir könnten uns darauf einigen, dass der Vergleich immer dann ohne Beachtung der Groß-/Kleinschreibung durchzuführen ist, wenn mindestens eines der zu vergleichenden Objekte die Groß-/Kleinschreibung ebenfalls nicht beachtet. Wenn wir nun aber das Beispiel ein wenig abändern und eine weitere Instanz von `basic_string` ins Spiel bringen, die eine dritte Vergleichsmethode mitbringt, ist die Sache nicht mehr so klar:

   ```
   typedef basic_string<char, yz_char_traits> yz_string;

   ci_string b = "aAa";
   yz_string c = "AAa";
   if( b == c ) /* ... */
   ```

   Wieder stellt sich die Frage, welches Ergebnis der Ausdruck »`b == c`« liefern soll. Sie stimmen sicher zu, dass es in diesem Fall weniger offensichtlich ist, welche Vergleichsmethode zu bevorzugen ist.

   Betrachten wir nun, wie viel klarer die Beispiele werden, wenn man sie wie folgt schreibt:

   ```
   string a = "aaa";
   string b = "aAa";
   if( stricmp( a.c_str(), b.c_str() ) == 0 ) /* ... */
   string c = "AAa";
   if( EqualUsingYZComparison( b, c ) ) /* ... */
   ```

   Es ist in vielen Fällen sinnvoller, die Unabhängigkeit von der Groß-/Kleinschreibung zu einer Eigenschaft der Vergleichsoperation zu machen. Ich bin allerdings in der Praxis auch Fällen begegnet, in denen es nützlicher war, diese Eigenschaft auf die Objekte zu verlagern (insbesondere, wenn die meisten oder alle Vergleiche mit C-Strings durchgeführt wurden), da Werte dann auf »natürliche« Weise verglichen

werden können (zum Beispiel »`if( a == "text" ) ...`«), ohne dass man sich jedes Mal daran zu erinnern hat, dass spezielle Vergleichsfunktionen zu benutzen sind.

Diese Lektion sollte Ihnen ein Gefühl dafür geben, wie das Template `basic_string` funktioniert und wie flexibel es in der Praxis ist. Wenn Sie andere Vergleiche wünschen, als mit `memicmp()` und `toupper()` möglich ist, ersetzen Sie einfach die hier gezeigten fünf Funktionen durch eigenen Code und führen damit Zeichenvergleiche durch, die zu Ihrer Anwendung passen.

Lektion 3: Von der Groß-/Kleinschreibung unabhängige Strings – Teil 2 Schwierigkeitsgrad: 5

Wie brauchbar ist der `ci_string`, den wir in Lektion 2 schufen? Im Folgenden befassen wir uns mit den Aspekten der Benutzbarkeit, beschäftigen uns mit auftretenden Entwurfsproblemen und -kompromissen und schließen einige der verbliebenen Lücken.

Betrachten wir wieder die Lösung aus Lektion 2 (ohne die Funktionsrümpfe):

```
struct ci_char_traits : public char_traits<char>
{
  static bool eq( char c1, char c2 )   { /*...*/ }
  static bool lt( char c1, char c2 )   { /*...*/ }
  static int compare( const char* s1,
                      const char* s2,
                      size_t n )       { /*...*/ }
  static const char*
  find( const char* s, int n, char a ) { /*...*/ }
};
```

Beantworten Sie für diese Lektion die folgenden Fragen so vollständig wie möglich:

1. Ist es sicher, `ci_char_traits` in dieser Weise von `char_traits<char>` abzuleiten?
2. Warum lässt sich der folgende Code nicht übersetzen?

   ```
   ci_string s = "abc";
   cout << s << endl;
   ```

3. Wie sieht es mit den anderen Operatoren (zum Beispiel +, +=, =) aus? Lassen sich `string`s und `ci_string`s als Argumente mischen? Beispiel:

   ```
   string    a = "aaa";
   ci_string b = "bbb";
   string    c = a + b;
   ```

Lösung

Die Antworten auf die drei Fragen lauten wie folgt.

1. Ist es sicher, `ci_char_traits` in dieser Weise von `char_traits<char>` abzuleiten?

   Dem Liskov-Substitutionsprinzip (LSP) entsprechend sollte öffentliche Vererbung normalerweise ein IST-EIN / ARBEITET-WIE-EIN modellieren (siehe Lektionen 22 und 28). Dieser Fall ist allerdings eine der seltenen Ausnahmen des LSP, da es nicht beabsichtigt ist, `ci_char_traits` polymorph unter Verwendung eines Zeigers oder einer Referenz auf ein Objekt der Basisklasse `char_traits<char>` anzusprechen. Die Vererbung wird hier nur aus Bequemlichkeit benutzt (einige würden sogar sagen, aus Faulheit) und nicht im objektorientierten Sinn.

   Auf der anderen Seite trifft das LSP weiterhin zu: Es gilt zur Übersetzungszeit, wenn das abgeleitete Objekt WIE EIN Basisobjekt ARBEITEN muss, um die Anforderungen des `basic_string`-Templates zu erfüllen. Nathan Myers[3] fasste das in einem Newsgroup-Artikel wie folgt in Worte:

   > *In anderen Worten: LSP gilt, jedoch nur zur Übersetzungszeit und dann als Konvention, die wir »Anforderungsliste« nennen. Ich würde diesen Fall gerne unterscheiden; nennen wir ihn Generisches Liskov-Substitutionsprinzip (GLSP): Jeder Typ (oder jedes Template), der (oder das) als ein Template-Argument übergeben wird, sollte die Anforderungsliste für dieses Argument erfüllen.*
   >
   > *Für Klassen, die von Iterator-Tags oder Traits-Klassen abgeleitet sind, gilt daher das GLSP. Die klassischen LSP-Betrachtungen (zum Beispiel virtuelle Destruktoren usw.) können oder können nicht zutreffen, je nach dem, ob die Anforderungsliste ein polymorphes Verhalten zur Laufzeit verlangt oder nicht.*

   Kurz gesagt, diese Ableitung ist sicher, da sie zum GLSP konform geht (wenn nicht sogar zum LSP). Trotzdem habe ich sie hier nicht aus Bequemlichkeit benutzt (um nicht ständig das `char_traits<char>` mitschleppen zu müssen), sondern um den *Unterschied* zu demonstrieren – dass wir nämlich zum Erreichen des gewünschten Effekts nur vier Operationen ändern mussten.

   Der Sinn der Frage besteht darin, Sie dazu zu bringen, über mehrere Dinge nachzudenken: (1) die korrekte Anwendung (und entsprechend der Missbrauch) von Vererbung, (2) die Folgen der Tatsache, dass es nur statische Klassenelemente gibt, (3) der Umstand, dass `char_traits`-Objekte niemals polymorph benutzt werden.

2. Warum lässt sich der folgende Code nicht übersetzen?

```
ci_string s = "abc";
cout << s << endl;
```

3. Nathan ist langjähriges Mitglied des C++-Standardisierungskomitees und anfänglicher Autor des `locale`-Teils des Standards.

Hinweis: Abschnitt 21.3.7.9 [lib.string.io] des C++-Standards spezifiziert die Deklaration von operator<< für basic_string so:

```
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const basic_string<charT,traits,Allocator>& str);
```

Antwort: Zuerst einmal ist cout offensichtlich vom Typ basic_ostream<char, char_traits<char> >. Des Weiteren ist der operator<< von basic_string als Template ausgeführt, wobei die Spezifikation nur das Einfügen in ein basic_ostream mit dem gleichen »Zeichentyp« und »Traits-Typ« wie bei string erlaubt. Das bedeutet, der Standardoperator operator<< ermöglicht die Ausgabe eines ci_string in ein basic_ostream<char, ci_char_traits>, was jedoch nicht dem Typ von cout entspricht, obwohl in der obigen Lösung ci_char_traits von char_traits<char> abgeleitet wurde.

Zur Lösung dieses Problems gibt es zwei Möglichkeiten: Definieren Sie für ci_strings Einfügen (operator<<()) und Entnahme (operator>>()) selbst oder hängen Sie ».c_str()« an, um so operator<<(const char*) benutzen zu können:

```
cout << s.c_str() << endl;
```

3. Wie sieht es mit den anderen Operatoren (zum Beispiel +, +=, =) aus? Lassen sich strings und ci_strings als Argumente mischen? Beispiel:

```
string    a = "aaa";
ci_string b = "bbb";
string    c = a + b;
```

Hier gibt es wieder zwei Möglichkeiten: Entweder Sie definieren Ihre eigene operator+()-Funktion, oder Sie fügen ».c_str()« hinzu, um so operator+(const char*) zu nutzen:

```
string    c = a + b.c_str();
```

Lektion 4: Maximal wiederverwendbare generische Container – Teil 1

Schwierigkeitsgrad: 8

Wie flexibel können Sie diese einfache Containerklasse machen? Hinweis: Sie werden nebenbei so einiges über Member-Templates lernen.

Wie lassen sich für die folgende, auf eine feste Länge ausgelegte Vektorklasse der Copy-Konstruktor und der Copy-Zuweisungsoperator am besten implementieren? Wie kann man Konstruktion und Zuweisung am universellsten auslegen? Hinweis: Denken Sie darüber nach, wie ein die Klasse anwendender Code aussehen könnte.

```
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;
  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T v_[size];
};
```

Beheben Sie bitte keine anderen Fehler. Dieser Container soll gar nicht vollständig STL-kompatibel sein und hat zudem mindestens ein subtiles Problem. Dieses Beispiel soll lediglich einige wichtige Fragestellungen in einer vereinfachten Umgebung beleuchten.

Lösung

Zur Lösung dieser Lektion gehen wir etwas anders vor. Ich werde eine Lösung vorschlagen, und Ihre Aufgabe besteht anschließend darin, dafür die fehlende Erklärung beizusteuern sowie die Lösung kritisch zu beurteilen. Wenden wir uns nun Lektion 5 zu.

Lektion 5: Maximal wiederverwendbare generische Container – Teil 2

Schwierigkeitsgrad: 6

Historische Anmerkung: Das in dieser Lektion benutzte Beispiel ist von einem Beispiel abgeleitet, das in den Ausgaben 12 und 20 des britischen C++-Magazins Overload von Kevlin Henney vorgestellt und später von Jon Jagger analysiert wurde. (Die britischen Leser werden schnell bemerken, dass die Lösung dieser Lektion weit über das hinausgeht, was in Overload, Ausgabe 20 gezeigt wurde. Tatsächlich funktioniert die dort vorgestellte Effizienzoptimierung nicht in der Lösung dieses Problems.)

Was geschieht in der folgenden Lösung, und warum? Erklären Sie jeden Konstruktor und Operator. Haben dieser Entwurf oder der Code irgendwelche Schwachstellen?

```
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;
```

```
  fixed_vector() { }

  template<typename O, size_t osize>
  fixed_vector( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
  }

  template<typename O, size_t osize>
  fixed_vector<T,size>&
  operator=( const fixed_vector<O,osize>& other )
  {
    copy( other.begin(),
          other.begin()+min(size,osize),
          begin() );
    return *this;
  }

  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T v_[size];
};
```


Lösung

Lassen Sie uns nun die obige Lösung daraufhin untersuchen, wie gut sie dem in der Frage gestellten Anspruch gerecht wird. Erinnern wir uns an die ursprünglichen Fragestellungen: Wie lassen sich für die folgende, auf eine feste Länge ausgelegte Vektorklasse der Copy-Konstruktor und der Copy-Zuweisungsoperator am besten implementieren? Wie kann man Konstruktion und Zuweisung am universellsten auslegen? Hinweis: Denken Sie darüber nach, wie ein die Klasse anwendender Code aussehen könnte.

Copy-Konstruktor und Copy-Zuweisungsoperator

Haben Sie bemerkt, dass die erste Frage nur ein Ablenkungsmanöver ist? Der ursprüngliche Code hat bereits einen Copy-Konstruktor und einen Copy-Zuweisungsoperator, und beide funktionieren hervorragend. Unser Lösungsvorschlag ist auf die zweite Frage ausgerichtet und fügt einen Template-Konstruktor sowie einen Template-Zuweisungsoperator hinzu, damit Konstruktion und Zuweisung flexibler werden.

```
template<typename O, size_t osize>
fixed_vector( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
}

template<typename O, size_t osize>
fixed_vector<T,size>&
operator=( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
  return *this;
}
```

Beachten Sie, dass es sich bei den obigen Funktionen nicht um einen Copy-Konstruktor und einen Copy-Zuweisungsoperator handelt, denn diese konstruieren bzw. kopieren nur aus anderen Objekten mit exakt dem gleichen Typ – was bei Template-Klassen eben auch die Template-Parameter einschließt. Ein Beispiel:

```
struct X
{
  template<typename T>
  X( const T& );    // KEIN Copy-Konstruktor,
                    // T kann nicht X sein
  template<typename T>
  operator=( const T& );
                    // KEINE Copy-Zuweisung, T kann nicht X sein
};
```

»Aber«, mögen Sie jetzt sagen, »diese zwei Template-Elementfunktionen könnten genau mit dem Copy-Konstruktor und dem Copy-Zuweisungsoperator übereinstimmen!« Nun, im Grunde ist dies in beiden Fällen unmöglich, denn `T` kann niemals `X` sein. Oder mit den Worten des Standards (12.8/2, *note* 4):

> *Da* ein Template-Konstruktor niemals ein Copy-Konstruktor ist, *unterdrückt das Vorhandensein eines solchen Template-Konstruktors nicht die implizite Deklaration eines Copy-Konstruktors. Template-Konstruktoren werden bei der Auflösung der Benutzung von überladenen Funktionen genauso berücksichtigt wie andere Konstruktoren, einschließlich des Copy-Konstruktors. Ein Template-Konstruktor könnte also zum Kopieren eines Objektes herangezogen werden, wenn durch ihn eine bessere Übereinstimmung erzielt werden kann als durch andere Konstruktoren.*

Eine ähnliche Erklärung existiert für den Copy-Zuweisungsoperator (12.8/9 *note* 7). Tatsächlich hat also die vorgeschlagene Lösung immer noch den gleichen Copy-Konstruktor und Copy-Zuweisungsoperator wie der ursprüngliche Code, da der Compiler

nach wie vor die impliziten Versionen erzeugt. Wir haben lediglich die Möglichkeiten zum Konstruieren und Zuweisen erweitert und nicht die alten Versionen ersetzt.

Betrachten wir für ein weiteres Beispiel das folgende Programm:

```
fixed_vector<char,4> v;
fixed_vector<int,4>  w;
fixed_vector<int,4>  w2(w);
        // ruft impliziten Copy-Konstruktor auf
fixed_vector<int,4>  w3(v);
        // ruft Template-Umwandlungskonstruktor auf
w = w2; // ruft impliziten Copy-Zuweisungsoperator auf
w = v;  // ruft Template-Zuweisungsoperator auf
```

Die Frage zielte also wirklich auf flexible »Konstruktion und Zuweisung durch andere `fixed_vectors`« und nicht speziell auf »Copy-Konstruktion und Copy-Zuweisung«, die es ja bereits gab.

Einige Aspekte beim Benutzen von Konstruktion und Zuweisung

Bezüglich der Verwendbarkeit sind hauptsächlich noch zwei Überlegungen anzustellen:

1. Unterstützung unterschiedlicher Typen (einschließlich Ableitungen)

   Obwohl `fixed_vector` definitiv ein homogener Container ist und auch bleiben soll, macht es zuweilen Sinn, eine entsprechende Instanz aus einem anderen `fixed_vector` zu konstruieren oder zuzuweisen, der Objekte anderen Typs enthält. Solange sich diese anderen Quellobjekte unserem Objekttyp zuweisen lassen, sollte dies erlaubt sein. Ein Anwender könnte zum Beispiel so etwas schreiben wollen:

   ```
   fixed_vector<char,4> v;
   fixed_vector<int,4>  w(v);  // Template-Konstruktion
   w = v;                      // Template-Zuweisung

   class B            { /*...*/ };
   class D : public B { /*...*/ };

   fixed_vector<D*,4> x;
   fixed_vector<B*,4> y(x);    // Template-Konstruktion
   y = x;                      // Template-Zuweisung
   ```

   Dies ist gestattet und funktioniert auch wie erwartet, da `D*` `B*` zugewiesen werden kann.

2. Unterstützung unterschiedlicher Größen

   In ähnlicher Weise könnte ein Anwender aus `fixed_vector`-Instanzen mit anderen Größen konstruieren oder zuweisen wollen. Wieder macht es Sinn, diese Fähigkeit zu unterstützen, zum Beispiel:

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);  // Initialisierung durch 4 Werte
w = v;                      // weist 4 Werte zu
class B            { /*...*/ };
class D : public B { /*...*/ };
fixed_vector<D*,16> x;
fixed_vector<B*,42> y(x);   // Initialisierung durch 16 Werte
y = x;                      // weist 16 Werte zu
```

Der alternative Standardbibliotheksansatz

Irgendwie gefallen mir Syntax und Anwendung der vorherigen Funktionen, es gibt allerdings noch einige praktische Dinge, die sich damit nicht durchführen lassen. Sehen wir uns einen anderen Ansatz an, der dem Stil der Standardbibliothek folgt:

1. Kopieren

```
template<class RAIter>
fixed_vector( RAIter first, RAIter last )
{
  copy( first,
first+min(size,(size_t)last-first),
begin() );
}
```

   Statt wie bisher beim Kopieren zu schreiben

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v);  // Initialisierung durch 4 Werte
```

   schreiben wir jetzt

```
fixed_vector<char,6> v;
fixed_vector<int,4>  w(v.begin(), v.end());
                            // Initialisierung durch 4 Werte
```

   Halten Sie einen Moment inne und denken Sie darüber nach. Was halten Sie davon? Welcher Stil ist für die Konstruktion besser geeignet – der Stil unserer vorgeschlagenen Lösung oder der der Standardbibliothek?

   In diesem Fall ist unsere anfänglich vorgeschlagene Lösung ein wenig einfacher zu benutzen, während der Ansatz im Stil der Standardbibliothek sehr viel flexibler ist (er erlaubt dem Benutzer zum Beispiel, Unterbereiche auszuwählen oder von ande-

ren Container-Arten zu kopieren). Sie können sich eine Methode aussuchen oder einfach beide Verhalten unterstützen.

2. Zuweisung

   Da `operator=()` nur einen Parameter haben kann, ist es hier unmöglich, in gleicher Weise einen Iterator-Bereich übergeben zu lassen. Stattdessen kann man dafür eine normale Funktion vorsehen:

```
template<class Iter>
fixed_vector<T,size>&
assign( Iter first, Iter last )
{
  copy( first,
        first+min(size,(size_t)last-first),
        begin() );
  return *this;
}
```

   An Stelle von

```
w = v;                          // Zuweisung von 4 Werten
```

   schreibt man jetzt bei der Zuweisung

```
w.assign(v.begin(), v.end());
                                // Zuweisung von 4 Werten
```

   Technisch gesehen ist `assign()` gar nicht notwendig, da sich die gleiche Flexibilität auch wie folgt erreichen lässt, allerdings weniger schön und effizient:

```
w = fixed_vector<int,4>(v.begin(), v.end());
    // Initialisiere und weise 4 Werte zu
```

   Halten Sie wieder einen Moment inne und denken Sie darüber nach. Welcher Stil ist für die Zuweisung besser geeignet – der Stil unserer vorgeschlagenen Lösung oder der der Standardbibliothek?

   Diesmal greift das Flexibilitätsargument nicht, da der Anwender die Kopie leicht selbst (und sogar flexibler) erstellen kann. Statt

```
w.assign( v.begin(), v.end() );
```

   schreibt er einfach

```
copy( v.begin(), v.end(), w.begin() );
```

   Ein spezielles `assign()` hat also in diesem Fall kaum eine Daseinsberechtigung. Für die Zuweisung ist es wahrscheinlich das beste, die Methode der vorgeschlagenen Lösung zu benutzen und den Anwender `copy()` benutzen zu lassen, wenn bereichsweise Zuweisung gefordert wird.

Wozu einen Default-Konstruktor schreiben?

Schließlich bleibt noch die Frage, warum in der vorgeschlagenen Lösung ein leerer Default-Konstruktor enthalten ist, der auch nur das Gleiche macht wie ein vom Compiler erzeugter Default-Konstruktor.

Die Antwort ist kurz und bündig: Es ist deshalb nötig, weil der Compiler in dem Moment keinen Default-Konstruktor mehr generiert, in dem man irgendeinen beliebigen Konstruktor definiert. Und Code wie der oben gezeigte braucht nun mal einen Default-Konstruktor.

Ein verbleibendes Problem

Oder auch: Hat dieses Design oder dieser Code irgendwelche Schwachstellen?

Leider ja! Ist Ihnen aufgefallen, dass der Template-Zuweisungsoperator nicht ganz exception-sicher ist? Erinnern Sie sich, wie er definiert war:

```
template<typename O, size_t osize>
fixed_vector<T,size>&
operator=( const fixed_vector<O,osize>& other )
{
  copy( other.begin(),
        other.begin()+min(size,osize),
        begin() );
  return *this;
}
```

Wenn eine der `T`-Zuweisungen während der Ausführung von `copy()` fehlschlägt, verbleibt das Objekt in einem inkonsistenten Zustand. Ein Teil des Inhalts unseres `fixed_vector`-Objekts wird unverändert bleiben, während der Rest bereits aktualisiert sein wird.

Leider kann `fixed_vector` mit dem momentanen Design *für Zuweisungen nicht exakt exception-sicher* gemacht werden. Warum nicht? Deshalb:

- Die korrekte (und einfachste) Methode, um dieses Problem zu lösen, besteht normalerweise darin, eine atomare `Swap()`-Funktion zur Verfügung zu stellen, die keine Exceptions auswirft und die die Innereien zweier `fixed_vector`-Instanzen austauscht. Diese wird dann innerhalb der kanonischen (und diesmal als Template ausgeführten) Form von `operator=()` benutzt, welche das »Temporäre-Variable-erzeugen-und-austauschen-Idiom« anwendet. Mehr dazu in Lektion 13.
- Es ist unmöglich, eine atomare, keine Exception werfende Funktion `swap()` zum Austausch der Internas zweier `fixed_vector`-Objekte zu schreiben, denn die `fixed_vector`-Daten sind als einfaches Array gespeichert, das nicht atomar kopiert werden kann. Genau das ist ja gerade das Problem des von uns vorgeschlagenen Zuweisungsoperators.

Aber noch besteht kein Grund zur Verzweiflung. Es gibt tatsächlich eine gute Lösung, allerdings muss dazu das `fixed_vector`-Design ein wenig geändert werden, so dass die Daten in einem dynamisch allozierten Array anstatt in einem Element-Array gespeichert werden. Natürlich geht dadurch der Vorteil der Effizienz verloren, und außerdem müssen wir nun einen Destruktor schreiben – aber so etwas kann der Preis für eine strenge Exception-Sicherheit sein.

```
// Eine streng exception-sichere Version:
//
template<typename T, size_t size>
class fixed_vector
{
public:
  typedef T*       iterator;
  typedef const T* const_iterator;

  fixed_vector() : v_( new T[size] ) { }

  ~fixed_vector() { delete[] v_; }

  template<typename O, size_t osize>
  fixed_vector( const fixed_vector<O,osize>& other )
    : v_( new T[size] )
    { copy( other.begin(), other.begin()+min(size,osize), begin() ); }
  fixed_vector( const fixed_vector<T,size>& other )
    : v_( new T[size] )
    { copy( other.begin(), other.begin()+min(size,osize), begin() ); }

  void Swap( fixed_vector<T,size>& other ) throw()
  {
    swap( v_, other.v_ );
  }

  template<typename O, size_t osize>
  fixed_vector<T,size>& operator=( const fixed_vector<O,osize>& other ) {
    fixed_vector<T,size> temp( other ); // erledigt alles
    Swap( temp ); return *this; // kein throw
  }
  fixed_vector<T,size>& operator=( const fixed_vector<T,size>& other ) {
    fixed_vector<T,size> temp( other ); // erledigt alles
    Swap( temp ); return *this; // kein throw
  }

  iterator       begin()       { return v_; }
  iterator       end()         { return v_+size; }
  const_iterator begin() const { return v_; }
  const_iterator end()   const { return v_+size; }

private:
  T* v_;
};
```

☒ Typischer Fehler

Verschieben Sie die Betrachtung der Exception-Sicherheit nie auf später. Exception-Sicherheit beeinflusst bereits den Entwurf einer Klasse und ist nicht lediglich »nur ein Implementierungsdetail«.

Zusammenfassung: Diese Lektion hat Sie hoffentlich davon überzeugt, dass Template-Elementfunktionen praktisch sind, und Ihnen gezeigt, warum sie in der Standardbibliothek so häufig benutzt werden. Verzweifeln Sie nicht, wenn Sie mit dieser Thematik nicht vertraut sind. Nicht alle Compiler unterstützen derzeit Template-Elementfunktion, werden es allerdings bald tun, da sie Bestandteil des Standards sind.

Setzen Sie in Ihren eigenen Klassen Member-Templates sinnvoll ein, und Sie werden wahrscheinlich nicht nur glückliche Anwender haben, sondern auch viel mehr Anwender, denn diese verwenden im Allgemeinen den Code, der sich am besten wiederverwenden lässt.

Lektion 6: Temporäre Objekte — Schwierigkeitsgrad: 5

Unnötige und/oder temporäre Objekte sind immer wieder daran schuld, dass die Ergebnisse all Ihrer harten Arbeit – und die Performance Ihres Programms – zum Fenster hinausgeworfen werden. Wie können Sie sie aufspüren und vermeiden?

(»Temporäre Objekte« könnten Sie einwenden, »sind doch eher ein Optimierungsproblem. Was hat das mit generischer Programmierung und der Standardbibliothek zu tun?« Haben Sie Geduld mit mir, der Grund wird in der folgenden Lektion klar werden.

Sie untersuchen einen fremden Code. Ein Programmierer hat die folgende Funktion geschrieben, in der an mindestens drei Stellen unnötige temporäre Objekte benutzt werden. Wie viele dieser Stellen können Sie identifizieren und welche Verbesserungen sollte der Programmierer dann jeweils vornehmen?

```
string FindAddr( list<Employee> emps, string name )
 {
   for( list<Employee>::iterator i = emps.begin();
        i != emps.end();
        i++ )
   {
     if( *i == name )
     {
       return i->addr;
     }
   }
   return "";
 }
```

Ändern Sie nicht die funktionelle Bedeutung dieser Funktion, auch wenn hier Verbesserungen möglich wären.

Lösung

Ob Sie es glauben oder nicht, in dieser kurzen Funktion verstecken sich drei offensichtliche Fälle unnötiger temporärer Objekte, zwei subtile und zwei falsche Spuren.

Die zwei offensichtlicheren temporären Objekte treten bereits in der Funktionsdeklaration auf:

```
string FindAddr(  list<Employee> emps , string name  )
```

Die Parameter sollten besser per `const&`-Referenz – also als `const list<Employee>&` bzw. `const string&` – übergeben werden, anstatt per Wert. Übergabe per Wert zwingt den Compiler dazu, vollständige Kopien beider Objekte zu machen. Das kann sehr aufwändig sein und ist hier wirklich unnötig.

☑ Richtlinie

Ziehen Sie die Objektübergabe per `const&`-Referenz der Übergabe per Wert vor.

Die dritte, offensichtlichere temporäre Variable tritt in der Abbruchbedingung der `for`-Schleife auf:

```
for( /*...*/ ;  i != emps.end() ; /*...*/ )
```

Bei den meisten Containern (einschließlich `list`) gibt `end()` ein temporäres Objekt zurück, das konstruiert und zerstört werden muss. Da sich der Wert nicht ändert, ist die wiederholte Berechnung (und somit Erzeugung und Zerstörung) in jedem Schleifenumlauf sowohl unnötig ineffizient als auch unästhetisch. Der Wert sollte nur einmal berechnet und dann zur Wiederverwendung in einer lokalen Variable gespeichert werden.

☑ Richtlinie

Berechnen Sie konstante Werte vorher, anstatt die entsprechenden Objekte wiederholt zu erzeugen.

Betrachten Sie als Nächstes die Art und Weise, in der in der `for`-Schleife `i` inkrementiert wird:

```
for( /*...*/ ;  i++  )
```

Diese temporäre Variable ist eine subtilere Angelegenheit, aber leicht zu verstehen, wenn Sie sich die Unterschiede zwischen Preinkrement und Postinkrement in Erinnerung rufen. Postinkrement ist gewöhnlich weniger effizient als Preinkrement, da der alte Wert zwischengespeichert und zurückgegeben werden muss. Die Implementierung von Postinkrement für eine Klasse `T` sollte generell in der kanonischen Form erfolgen:

```
const T T::operator++(int)()
 {
   T old( *this ); // ursprünglichen Wert sichern
   ++*this;        // Postinkrement immer durch
                   // Preinkrement ausdrücken
   return old;     // ursprünglichen Wert zurückgeben
 }
```

Jetzt ist es leicht zu sehen, warum Postinkrement weniger effizient als Preinkrement ist. Postinkrement muss die gleiche Arbeit wie Preinkrement erledigen, hat aber zusätzlich noch ein anderes Objekt zu konstruieren und zurückzugeben, das den alten Wert enthält.

☑ Richtlinie

Implementieren Sie zur Wahrung der Konsistenz Postinkrement immer durch Preinkrement, andernfalls erhalten Ihre Anwender überraschende (und oft unerfreuliche) Ergebnisse.

Im gezeigten Code wird der Originalwert niemals gebraucht, daher gibt es keine Rechtfertigung für Postinkrement. Besser ist hier Preinkrement. Im Kasten »Wann können Compiler Postinkrement optimieren?« wird beschrieben, warum Compiler diesen Austausch im Allgemeinen nicht automatisch ausführen können.

☑ Richtlinie

Bevorzugen Sie Preinkrement. Nutzen Sie Postinkrement nur dann, wenn Sie den ursprünglichen Wert noch brauchen.

```
if(  *i == name  )
```

Die Klasse `Employee` wird in der Problemstellung nicht näher spezifiziert, wir können jedoch trotzdem einige Folgerungen ableiten. Damit der Code überhaupt funktioniert, verfügt `Employee` wahrscheinlich über eine Funktion zur Umwandlung in `string` oder über einen Umwandlungskonstruktor mit `string` als Argument. In beiden Fällen wird hier ein temporäres Objekt erzeugt und anschließend `operator==()` für `string`s oder `ope-`

rator==() für Employees aufgerufen. Ein temporäres Objekt wird nur dann nicht benötigt, wenn es einen operator==() gibt, der Objekte beider Typen vergleichen kann, oder wenn Employee über eine Umwandlung in eine Referenz, also string&, verfügt.

☑ Richtlinie

Achten Sie auf versteckte temporäre Objekte, die durch implizite Typumwandlungen entstehen. Eine gute Methode, das zu vermeiden, besteht darin, Konstruktoren wenn möglich `explicit` *zu machen und Typumwandlungsoperatoren zu vermeiden.*

Wann können Compiler Postinkrement optimieren?

Gesetzt den Fall, Sie schreiben einen Postinkrementausdruck wie »i++«, benutzen jedoch den Rückgabewert nicht. Darf der Compiler dann zur Optimierung den Postinkrementausdruck durch Preinkrement ersetzen?

Die Antwort darauf lautet: Nein, im Allgemeinen nicht. Nur bei eingebauten Datentypen und bei Standardtypen, also zum Beispiel bei int und complex, bei denen gewisse Annahmen über deren Bedeutung zulässig sind, da sie zum Standard gehören, hat der Compiler die Möglichkeit, Postinkrement wegzuoptimieren, indem er es durch Preinkrement ersetzt.

Bei beliebigen Klassen ist dem Compiler die tatsächliche Bedeutung von Pre- und Postinkrement nicht bekannt – in der Tat könnten damit verschiedene Dinge gemeint sein. Natürlich wäre es boshaft, wenn diese beiden Funktionen nicht die gleiche Bedeutung hätten, und jeder Programmierer, der so etwas programmiert, sollte auf der Stelle rausgeschmissen werden. Dem Compiler hilft das freilich wenig, da er keinen konsistenten Programmierer voraussetzen kann.

Es gibt eine Möglichkeit, dem Compiler die Beziehung zwischen Pre- und Postinkrement für eine Klasse mitzuteilen: Implementieren Sie Postinkrement in der kanonischen Form, d.h. rufen Sie darin Preinkrement auf, und deklarieren Sie es inline, so dass der Compiler über den Funktionsaufruf hinweg die Definition einsehen (sofern er die inline-Direktive respektiert) und das unbenutzte temporäre Objekt erkennen kann. Ich empfehle das allerdings nicht, denn inline ist kein Allheilmittel, kann vom Compiler ignoriert werden und bewirkt unter anderem eine stärkere Kopplung. Die weitaus einfachere Lösung besteht darin, sich einfach die Benutzung von Preinkrement für die Fälle anzugewöhnen, in denen der ursprüngliche Wert nicht gebraucht wird. Die oben genannte Optimierung wird dann niemals notwendig werden.

```
return  i->addr;
return  "";
```

Hier handelt es sich um eine subtile falsche Spur. Natürlich erzeugen beide Anweisungen temporäre `string`-Objekte, aber diese Objekte können nicht vermieden werden.

In der Vergangenheit habe ich oft die Argumentation gehört, es sei besser, ein lokales `string`-Objekt zu deklarieren, das den Rückgabewert aufnimmt, und diesen `string` mit einer einzigen `return`-Anweisung zurückzugeben (`string ret; ... ret = i->addr; break; ... return ret;`). Nun wird zwar diese Single-Entry/Single-Exit-Vorgehensweise (SE/SE) den Code häufig lesbarer (und manchmal schneller) machen, ob damit allerdings Verbesserungen oder Verschlechterungen der Performance erreicht werden, kann sehr stark vom tatsächlichen Code und vom Compiler abhängen.

In diesem Fall besteht das Problem darin, dass die Erzeugung eines lokalen `string`-Objektes und die folgende Zuweisung an dasselbe den Aufruf sowohl des Default-Konstruktors als auch des Zuweisungsoperators von `string` erfordert, im Gegensatz zum einfachen Konstruktoraufruf unseres ursprünglichen Codes. »Aber,« werden Sie jetzt fragen, »wie aufwändig kann der simple Default-Konstruktor von `string` schon sein?« Nun, die Version mit den zwei `return`s verhält sich unter Verwendung eines populären Compilers folgerndermaßen:

- Mit abgeschalteter Optimierung: 5% schneller als die Version mit dem lokalen `string`-Objekt
- Mit aggressiven Optimierungen: 40% schneller als die Version mit dem lokalen `string`-Objekt

```
string FindAddr( /*...*/ )
```

Das ist die zweite falsche Spur. Es scheint so, als könnte man das temporäre Objekt bei allen Rückgaben vermeiden, indem man den Rückgabetyp in eine Refernz abändert, hier also `string` durch `string&` ersetzt. Falsch! (Jedenfalls im Allgemeinen; siehe auch nächste Richtlinie.) Falls Sie Glück haben, stürzt Ihr Programm in dem Moment ab, in dem es versucht, die Referenz zu dereferenzieren, da das zugehörige lokale Objekt nicht mehr existiert. Es kann aber auch passieren, dass der Code periodisch scheinbar funktioniert und dann wieder versagt, was Ihnen lange Nächte bescheren wird, in denen Sie sich mit dem Debugger herumplagen werden.

☑ Richtlinie

Achten Sie auf die Lebensdauer der Objekte. Geben Sie niemals Zeiger oder Referenzen auf lokale automatische Objekte zurück; sie sind völlig nutzlos, denn der aufrufende Code kann sie nicht dereferenzieren, könnte es aber (was noch schlimmer ist) versuchen.

Obwohl ich dem hier nicht weiter nachgehen werde, sei der Vollständigkeit halber erwähnt, dass es eine Möglichkeit gibt, in vertretbarer Weise eine Referenz zurückzugeben und so ein temporäres Objekt zu vermeiden. Kurz:

```
const string&
FindAddr( /* ... */ )
{
  for( /* ... */ )
  {
    if( /* gefunden */ )
    {
      return i->addr;
    }
  }
  static const string empty;
  return empty;
}
```

In der Dokumentation der Funktion muss jetzt natürlich die Lebensdauer der Referenz exakt definiert werden. Wird das Objekt gefunden, geben wir eine Referenz auf einen `string` innerhalb eines `Employee`-Objekts innerhalb eines `list`-Containers zurück, die nur so lange gültig ist wie das `Employee`-Objekt innerhalb des Containers selbst. Ferner kann sich der Wert des Strings ändern, wenn das zugehörige `Employee`-Objekt geändert wird.

Ich werde das hier nicht weiter verfolgen, da diese Verfahrensweise in der Praxis meist nicht viel einbringt. Außerdem können Programmierer in Bezug auf die Lebensdauer zurückgegebener Referenzen notorisch vergesslich und sorglos sein:

```
string& a = FindAddr( emps, "John Doe" );
emps.clear();
cout << a;  // Fehler
```

Wenn der anwendende Programmierer so oder ähnlich verfährt und eine Referenz auch nach Ablauf ihrer Lebensdauer benutzt, entsteht ein Fehler, der typischerweise unregelmäßig auftritt und schwer zu entdecken ist. In der Tat besteht einer der Fehler, die Programmierer bei Anwendung der Standardbibliothek am häufigsten machen, darin, Iteratoren zu benutzen, die nicht mehr gültig sind, was so ziemlich das Gleiche ist wie die Benutzung einer Referenz, deren Lebensdauer abgelaufen ist (siehe Lektion 1 bezüglich der versehentlichen Benutzung ungültiger Iteratoren).

Es gibt noch einige andere Optimierungsmöglichkeiten, die wir aber jetzt erst einmal ignorieren wollen. Stattdessen betrachten wir nun die korrigierte Version von `FindAddr`, die die unnötigen temporären Objekte vermeidet. Beachten Sie die Verwendung von `const_iterator`, die nötig wurde, da der Parameter `list<Employee>` jetzt `const` ist.

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator end( emps.end() );
  for( list<Employee>::const_iterator i = emps.begin();
       i != end;
       ++i )
  {
    if( i->name == name )
    {
      return i->addr;
    }
  }
  return "";
}
```

Lektion 7: Benutzen der Standardbibliothek (oder: Wiedersehen mit den temporären Objekten) Schwierigkeitsgrad: 5

Effektive Wiederverwendung ist ein wichtiger Bestandteil guten Software-Entwurfs. Sie stehen viel besser da, wenn Sie die Algorithmen der Standardbibliothek verwenden, anstatt Ihre eigenen anzufertigen. Lassen Sie uns, um das zu demonstrieren, noch einmal die vorangegangene Lektion betrachten und untersuchen, wie viele der Probleme durch Wiederverwendung dessen, was bereits in der Standardbibliothek zur Verfügung steht, hätten vermieden werden können.

Wie viele der Fallstricke der Lektion 6 hätte der Programmier umgehen können, wenn er die Schleife nicht selbst kodiert, sondern Algorithmen der Standardbibliothek verwendet hätte? Demonstrieren Sie es. (Und ändern Sie wie in Lektion 6 nicht die Bedeutung der Funktion, auch wenn Verbesserungen möglich wären.)

Zur Erinnerung hier noch einmal die am stärksten korrigierte Version:

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator end( emps.end() );
  for( list<Employee>::const_iterator i = emps.begin();
       i != end;
       ++i )
  {
    if( i->name == name )
    {
      return i->addr;
    }
  }
  return "";
}
```

Lösung

Ohne weitere Änderungen könnten zwei temporäre Objekte sowie die ineffiziente wiederholte Berechnung von `emps.end()` vermieden werden, wenn man einfach den Standard-`find()`-Algorithmus verwenden würde.

```
string FindAddr( list<Employee> emps, string name )
{
  list<Employee>::iterator i(
    find( emps.begin(), emps.end(), name )
    );
  if( i != emps.end() )
  {
    return i->addr;
  }
  return "";
}
```

Man kann das natürlich mit Funktoren und `find_if` noch weiter treiben, aber allein dieses einfache Wiederverwenden von `find` spart schon Programmieraufwand und erhöht die Laufzeiteffizienz.

☑ Richtlinie

Verwerten Sie Code – speziell den der Standardbibliothek – wieder, anstatt eigenen zu schreiben. Ihr Programm wird dadurch schneller, einfacher und sicherer.

Die Wiederverwendung existierenden Codes ist gewöhnlich besser als jede Eigenanfertigung. Die Standardbibliothek enthält jede Menge Code, der dazu geschaffen wurde, benutzt und wiederverwendet zu werden. Konsequenterweise wurde das Design dieser Bibliothek, einschließlich ihrer Algorithmen wie `find()` und `sort()`, mit großer Sorgfalt vorgenommen. Die für Implementierung zuständigen Experten diskutierten stundenlang selbst über die kleinsten Einzelheiten und befassten sich detailliert mit Aspekten der Effizienz und der Anwendbarkeit sowie einer Menge anderer Umstände, so dass Sie sich nicht mehr damit herumschlagen müssen. Nutzen Sie das aus, indem Sie Code, insbesondere den Code der Standardbibliothek, wiederverwenden.

In Kombination mit anderen Korrekturen erhalten wir nun eine erheblich verbesserte Funktion.

```
string FindAddr( const list<Employee>& emps,
                 const string&         name )
{
  list<Employee>::const_iterator i(
    find( emps.begin(), emps.end(), name )
```

```
        );
      if( i != emps.end() )
      {
        return i->addr;
      }
      return "";
    }
```

3 Exception-Sicherheit

Lassen Sie uns zu Beginn ein wenig die Geschichte dieses Themas betrachten. 1994 veröffentlichte Tom Cargill den folgenreichen Artikel »Exception Handling: A False Sense of Security« (Cargill94).[1] Er demonstrierte auf schlüssige Weise, dass es die C++-Gemeinschaft zu diesem Zeitpunkt noch nicht vollständig verstand, exception-sicheren Code zu schreiben. In der Tat war nicht einmal bekannt, welche Aspekte denn nun für die Exception-Sicherheit wichtig waren oder wie man überhaupt darüber vernünftig reden sollte. Cargill forderte jeden heraus, eine geschlossene Lösung für dieses Problem zu finden. Drei Jahre vergingen. Einige Leute reagierten auf Teilaspekte von Cargills Beispiel, aber niemandem gelang eine umfassende Lösung.

Schließlich erschien 1997 *Guru of the Week #8* in der Internet-Newsgruppe *comp.lang.c++.moderated*. Nummer 8 verursachte wochenlange Diskussionen und führte zur ersten vollständigen Lösung von Cargills Herausforderung. Später im gleichen Jahr wurde eine stark erweiterte Version in den September- und November/Dezember-Ausgaben des *C++ Report* unter dem Titel »Exception-Safe Generic Containers« veröffentlicht, die an die aktuellsten Änderungen des Entwurfs des C++-Standards angepasst war und nicht weniger als drei vollständige Lösungen demonstrierte. (Kopien dieser Originalartikel werden auch im in Kürze erscheinenden Buch *C++ Gems II* [Martin00] zu finden sein.)

Anfang 1999 nahm Scott Meyers eine Kombination dieser Artikel mit auf seine *Effective C++ CD* (Meyers99) auf, zusammen mit Cargills ursprünglicher Herausforderung sowie dem aktualisierten Text seiner Klassiker *Effective C++* und *More Effective C++*.

Diese Miniserie hat seit der ersten Veröffentlichung als *Guru of the Week #8* einen langen Weg zurückgelegt. Ich hoffe, Sie finden sowohl Gefallen daran als auch Nutzen darin. Besonderer Dank gebührt Dave Abrahams und Greg Colvin, ebenfalls Komiteemitglieder wie ich, für ihr Verständnis der Art und Weise, in der Exception-Sicherheit zu betrachten ist, sowie für ihre aufmerksamen Kritiken mehrerer Entwürfe dieses Materials. Dave und Greg sind zusammen mit Matt Austern die Autoren der zwei

1. »Exception-Behandlung: Ein falsches Gefühl der Sicherheit« (Anm. d. Übers.), in englischer Sprache online auf `http://cseng.awl.com/bookdetail.qry?ISBN=0-201-63371-X&ptype=636` abrufbar.

vollständigen Komiteevorschläge zur Einführung der aktuellen Exception-Sicherheitsgarantien in die Standardbibliothek.

Lektion 8: Entwurf exception-sicheren Codes – Teil 1 Schwierigkeitsgrad: 7

Exception-Behandlung und Templates sind zwei der mächtigsten C++-Merkmale. Exception-sicheren Code zu schreiben kann jedoch recht schwierig sein – besonders in einem Template, wenn man keinen Anhaltspunkt hat, welche Exception wann in einer bestimmten Funktion ausgeworfen wird.

Diese Miniserie befasst sich sowohl mit Exception-Behandlung als auch mit Templates, indem sie untersucht, welche Anforderungen an exception-sichere (korrekte Funktionsweise in Gegenwart von Exceptions) und exception-neutrale (alle Exceptions an den Aufrufer weiterleiten) generische Container zu stellen sind. Das ist zwar eine leicht zu formulierende, jedoch nicht zu unterschätzende Aufgabe.

Nun denn, auf geht's! Versuchen Sie, einen einfachen Container zu implementieren (einen `Stack`, der *push* und *pop* kennt) und lernen Sie die Probleme kennen, die dabei auftreten, wenn Sie ihn exception-sicher und exception-neutral machen.

Wir beginnen dort, wo Cargill aufgehört hat – nämlich indem wir schrittweise eine sichere Version des `Stack`-Templates entwerfen, das von ihm kritisiert wurde. Später werden wir den `Stack`-Container erheblich verbessern. Dazu reduzieren wir die Anforderungen an `T`, den enthaltenen Typ, und befassen uns mit fortschrittlichen Methoden zur exception-sicheren Ressourcenverwaltung. Sukzessive werden wir dann auch die Antworten auf Fragen wie diese erhalten:

- Was sind die verschiedenen »Niveaus« der Exception-Sicherheit?
- Können oder sollten generische Container vollkommen exception-neutral sein?
- Sind die Container der Standardbibliothek exception-sicher oder exception-neutral?
- Beeinflusst Exception-Sicherheit das Design der öffentlichen Schnittstelle eines Containers?
- Sollten generische Container von Exception-Spezifikationen Gebrauch machen?

Es folgt nun die Deklaration des `Stack`-Templates, das im Wesentlichen dem aus Cargills Artikel entspricht. Ihre Aufgabe lautet nun: Machen Sie `Stack` exception-sicher und exception-neutral, d.h. `Stack`-Objekte sollen sich unabhängig davon, ob während der Ausführung der `Stack`-Elementfunktionen irgendwelche Exceptions ausgeworfen werden, immer in einem korrekten und konsistenten Zustand befinden. Da nur der Aufrufer über `T` genauer Bescheid weiß, sollen alle Exceptions vollständig an ihn weitergeleitet werden.

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  /*...*/
private:
  T*     v_;      // Zeiger auf einen Speicherbereich
  size_t vsize_;  // groß genug für 'vsize_' T's
  size_t vused_;  // # der tatsächlich benutzten T's
};
```

Entwerfen Sie einen nachweislich exception-sicheren (funktioniert korrekt in Gegenwart von Exceptions) und exception-neutralen (leitet alle Exceptions an den Aufrufer weiter, ohne Integritätsprobleme zu verursachen) Default-Konstruktor und -Destruktor für diese Klasse.

Lösung

Man kann sofort erkennen, dass `Stack` dynamischen Speicher wird verwalten müssen. Somit ist die Vermeidung von Speicherlecks, auch in Gegenwart von Exceptions, die von Operationen mit `T` und von Speicherallozierungen ausgeworfen werden, ein wichtiger Aspekt. Fürs Erste belassen wir die Speicherverwaltung innerhalb der `Stack`-Elementfunktionen. Im weiteren Verlauf dieser Miniserie verlagern wir sie dann in eine private Basisklasse, um den Ressourcenbesitz zu kapseln.

Default-Konstruktion

Betrachten Sie einen möglichen Default-Konstruktor:

```
// Ist das sicher?
template<class T>
Stack<T>::Stack()
 : v_(0),
   vsize_(10),
   vused_(0)         // bis jetzt keine benutzt
{
  v_ = new T[vsize_]; // Anfangsallozierung
}
```

Ist dieser Konstruktor exception-sicher und exception-neutral? Um das herauszufinden, überlegen wir uns, wo Exceptions ausgeworfen werden können. Kurz gesagt: Jede Funktion könnte das. Der erste Schritt besteht also darin, festzustellen, welche Funktionen tatsächlich aufgerufen werden. Das schließt sowohl freie Funktionen als auch Konstruktoren, Destruktoren, Operatoren und andere Elementfunktionen ein.

Der `Stack`-Konstruktor setzt zuerst `vsize_` auf 10 und versucht dann, durch `new T[vsize_]` Speicher zu allozieren. Bei dieser letztgenannten Operation wird zunächst `operator new[]()` aufgerufen (entweder der standardmäßig vorgegebene `operator new[]()` oder ein von `T` zur Verfügung gestellter) und anschließend versucht, `T::T` `vsize_`-mal aufzurufen. Zwei Dinge können dabei schiefgehen, nämlich einerseits die Speicherallozierung selbst, wobei `operator new[]()` dann eine `bad_alloc`-Exception auswerfen würde, und andererseits der Default-Konstruktor von `T`, der letztlich jede mögliche Exception auswerfen könnte, wobei dann jedes bis dahin konstruierte Objekt wieder zerstört und der allozierte Speicher garantiert automatisch via `operator[]()` freigegeben werden würde.

Die obige Funktion ist daher vollständig exception-sicher und exception-neutral, und wir können uns dem nächsten Thema ... Wie bitte? Sie wollen wissen, warum die Funktion so robust ist? Na gut, lassen Sie uns die Sache etwas detaillierter untersuchen.

1. *Wir sind exception-neutral*. Wir fangen keine Exceptions auf, so dass alle Exceptions, die eventuell von `new` geworfen werden, direkt an den Aufrufer weitergeleitet werden.

☑ Richtlinie

Soll eine Funktion nicht mit einer Exception umgehen (oder sie umwandeln oder bewusst absorbieren), sollte sie sie zu einem Aufrufer durchlassen, der darauf korrekt reagieren kann.

2. *Wir verlieren keinen Speicher.* Würde der Aufruf von `operator new[]()` durch den Auswurf einer `bad_alloc`-Exception beendet werden, gäbe es gar keinen allozierten Speicher, ein Speicherleck kann daher nicht entstehen. Der Auswurf einer Exception durch den `T`-Konstruktor führt dazu, dass alle bis dahin konstruierten `T`-Objekte korrekt zerstört werden und schließlich `operator delete[]()` aufgerufen wird, was den Speicher freigibt. Das sichert uns wie angekündigt vor Speicherlecks.

   Ich werde im Folgenden nicht mehr darauf eingehen, dass einer der `T`-Destruktoren während der Aufräumarbeiten eine Exception auswerfen könnte. In einer solchen Situation würde `terminate()` aufgerufen und das Programm abgebrochen werden, was uns die Ablaufkontrolle aus der Hand nähme. Warum Destruktoren, die Exceptions werfen, »böse« sind, wird in Lektion 16 näher erläutert.

3. *Wir bleiben in einem konsistenten Zustand, unabhängig davon, ob irgendein Teil des `new` eine Exception wirft*. Nun könnte man einwenden, dass, wenn `new` eine Exception ausgeworfen hat, `vsize_` bereits auf 10 gesetzt ist, obwohl gar nichts erfolgreich alloziert werden konnte. Ist das nicht inkonsistent? Nicht wirklich, denn es ist irrelevant. Erinnern Sie sich, dass wir die von `new` geworfene Exception direkt an den

Aufrufer, außerhalb unseres Konstruktors, weiterleiten? Per Definition bedeutet das »Verlassen eines Konstruktors aufgrund einer Exception« für unser `Stack`-Proto-Objekt, dass es niemals zu einem vollständig konstruierten Objekt wird. Seine Lebenszeit beginnt gar nicht, es hat nie existiert, also ist sein Zustand ohne Bedeutung. Es spielt keine Rolle, welchen Wert der Speicher enthält, den `vsize_` für kurze Zeit belegte, genauso wenig wie es von Bedeutung ist, was der Speicher enthält, der nach einem Destruktoraufruf zurückbleibt. Bei all dem handelt es sich um nicht allozierten »Rohspeicher«, Asche und Rauch.

☑ Richtlinie

Strukturieren Sie Ihren Code immer so, dass auch in Gegenwart von Exceptions Ressourcen korrekt freigegeben werden und Daten in konsistentem Zustand bleiben.

Ok, ich gebe zu, dass ich das `new` nur deshalb im Konstruktorrumpf platziert habe, um diese dritte Diskussion führen zu können. Was ich eigentlich schreiben möchte, ist Folgendes:

```
template<class T>
Stack<T>::Stack()
  : v_(new T[10]),  // Default-Allozierung
    vsize_(10),
    vused_(0)       // bis jetzt keine benutzt
{
}
```

Beide Versionen sind so gut wie äquivalent. Ich bevorzuge die letztere, weil sie der üblichen Vorgehensweise folgt, Klassenelemente wenn möglich schon in der Initialisierungsliste zu initialisieren.

Zerstörung

Wenn wir erst einmal eine (überaus) vereinfachende Annahme machen, sieht der Destruktor schon viel einfacher aus.

```
template<class T>
Stack<T>::~Stack()
{
  delete[] v_;     // Dies wirft keine Exceptions aus
}
```

Warum kann `delete[]` keine Exceptions auswerfen? Rufen Sie sich ins Gedächtnis zurück, dass dies für jedes Objekt im Array den Aufruf von `T::~T` sowie den anschließenden Aufruf des `operator delete[]()` bedeutet. Wir wissen, dass die Speicherfreigabe

durch `operator delete[]()` niemals Exceptions auswirft, denn der Standard verlangt, dass dieser Operator eine der beiden folgenden Signaturen aufweist:[2]

```
void operator delete[]( void* ) throw();
void operator delete[]( void*, size_t ) throw();
```

Die einzig mögliche Quelle für Exceptions könnte daher nur noch einer der `T::~T`-Aufrufe sein, jedoch haben wir bereits willkürlich festgelegt, dass der `Stack`-Destruktor keine Exceptions auswirft. Warum? Um es kurz zu machen, wir können einfach keinen vollständig exception-sicheren Destruktor implementieren, wenn `T::~T` selbst Exceptions werfen darf. Dass `T::~T` keine Exceptions werfen darf, ist aber keine besonders große Einschränkung, denn es gibt jede Menge anderer Gründe, warum man Destruktoren nicht erlauben sollte, Exceptions auszuwerfen.[3] Jede Klasse, deren Destruktor Exceptions werfen kann, wird Ihnen früher oder später alle möglichen Sorten anderer Probleme einbringen. Außerdem können Sie einfach nicht verlässlich ein Array davon mit `new[]` und `delete[]` anlegen bzw. freigeben. Mehr dazu, wenn wir mit dieser Miniserie fortfahren.

☑ Richtlinie

Beachten Sie die kanonischen Exception-Sicherheitsregeln: Gestatten Sie niemals einer Exception, einen Destruktor oder einen überladenen `operator delete()` bzw. `operator delete[]()` zu verlassen. Schreiben Sie jeden Destruktor und jede Deallozierungsfunktion so, als hätten sie eine Exception-Spezifikation »`throw()`«. Mehr dazu im Folgenden; hierbei handelt es sich um ein wichtiges Thema.

Lektion 9: Entwurf exception-sicheren Codes – Teil 2 Schwierigkeitsgrad: 8

Wo wir jetzt den Default-Konstruktor und den Destruktor in der Tasche haben, könnten wir möglicherweise versucht sein zu denken, dass das mit den anderen Funktionen genauso klappt. Wie wir allerdings gleich sehen werden, halten der Copy-Konstruktor und der Zuweisungsoperator in Sachen Exception-Sicherheit und -Neutralität ihre eigenen Herausforderungen für uns bereit.

2. In einer privaten Unterhaltung wies Scott Meyers darauf hin, dass dies streng genommen niemanden davon abhält, einen überladenen `operator delete[]` zu schreiben, der doch Exceptions auswirft. Jede derartige Überladung würde allerdings jenen klaren Vorsatz verletzen und sollte daher als fehlerhaft bzw. unzulässig betrachtet werden.
3. Offen gestanden werden Sie nicht viel falsch machen, wenn Sie gewohnheitsmäßig hinter jede Ihrer Destruktordeklarationen `throw()` schreiben. Selbst wenn Exception-Spezifikationen bei Ihrem derzeitigen Compiler zu aufwändigen Prüfungen führen, sollten Sie wenigstens alle Ihre Destruktoren so schreiben, als wären sie als `throw()` spezifiziert – den Exceptions also niemals erlauben, jemals einen Destruktor zu verlassen.

Betrachten Sie wieder das `Stack`-Template von Cargill:

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  /*...*/
private:
  T*     v_;      // Zeiger auf einen Speicherbereich
  size_t vsize_;  // groß genug für 'vsize_' T's
  size_t vused_;  // # der tatsächlich benutzten T's
};
```

Lösung

Lassen Sie uns zur Implementierung des Copy-Konstruktors und des Copy-Zuweisungsoperators eine von beiden eingesetzte Funktion namens `NewCopy` benutzen, die die Allozierung und das Vergrößern des Speichers besorgt. `NewCopy` übernimmt einen Zeiger (`src`) auf einen vorhandenen `T`-Pufferspeicher sowie dessen Größe (`srcsize`) und gibt einen Zeiger auf einen neuen und möglicherweise größeren Puffer zurück. Die Verantwortung für diesen neuen Puffer tritt die Funktion an den Aufrufer ab. Bei einer Exception gibt `NewCopy` alle temporären Ressourcen frei und leitet die Exception weiter, so dass keine Lecks entstehen.

```
template<class T>
T* NewCopy( const T* src,
            size_t   srcsize,
            size_t   destsize )
{
  assert( destsize >= srcsize );
  T* dest = new T[destsize];
  try
  {
    copy( src, src+srcsize, dest );
  }
  catch(...)
  {
    delete[] dest; // kein throw
    throw;         // ursprüngliche Exception werfen
  }
  return dest;
}
```

Analysieren wir diesen Code nun schrittweise.

1. In der `new`-Anweisung kann entweder eine `bad_alloc`-Exception durch die Allozierung oder irgendeine Exception von `T::T` geworfen werden. In beiden Fällen wird nichts alloziert und die Exception einfach weitergeleitet. Das verursacht keine Lecks und ist exception-neutral.

2. Als Nächstes weisen wir alle vorhandenen Werte mit Hilfe von `T::operator=()` zu. Schlägt eine der Zuweisungen fehl, fangen wir die Exception auf, geben den allozierten Speicher frei und leiten die Exception weiter. Wieder verursachen wir keine Lecks und bleiben exception-neutral. Es gibt hier aber noch etwas zu beachten: `T::operator=()` muss garantieren, dass das Objekt, an das zugewiesen werden soll, beim Auftreten einer Exception unverändert bleibt.[4]

3. Wenn sowohl die Allozierung als auch das Kopieren erfolgreich verliefen, geben wir den Zeiger auf den neuen Puffer mitsamt der Verantwortung dafür zurück (von da an ist also der Aufrufer für den Puffer zuständig). Da mit `return` lediglich der Zeigerwert kopiert wird, kann hier keine Exception auftreten.

Copy-Konstruktion

Mit `NewCopy` gestaltet sich der Copy-Konstruktor von `Stack` einfach.

```
template<class T>
Stack<T>::Stack( const Stack<T>& other )
  : v_(NewCopy( other.v_,
                other.vsize_,
                other.vsize_ )),
    vsize_(other.vsize_),
    vused_(other.vused_)
{
}
```

Copy-Zuweisung

Als Nächstes gehen wir die Copy-Zuweisung an.

```
template<class T>
Stack<T>&
Stack<T>::operator=( const Stack<T>& other )
{
  if( this != &other )
  {
    T* v_new = NewCopy( other.v_,
```

4. Im weiteren Verlauf werden wir `Stack` so weiter entwickeln, dass wir nicht mehr auf `T::operator=()` angewiesen sind.

```
                        other.vsize_,
                        other.vsize_ );
      delete[] v_;  // kein throw
      v_ = v_new;   // Zuständigkeit übernehmen
      vsize_ = other.vsize_;
      vused_ = other.vused_;
    }
    return *this;   // sicher, da keine Kopie
  }
```

Wieder kann nach dem routinemäßigen schwachen Schutz gegen Selbstzuweisung nur `NewCopy` Exceptions auswerfen. Wenn das passiert, leiten wir die Exception korrekt weiter, ohne dass der Objektzustand davon betroffen wäre. Der Aufrufer sieht bei einer Exception, dass nichts verändert wurde, während ohne Exception die Zuweisung mit allen ihren Seiteneffekten durchgeführt wird.

Es wird hier das folgende sehr wichtige Exception-Sicherheits-Idiom deutlich.

☑ Richtlinie

Beachten Sie die kanonischen Exception-Sicherheitsregeln: Schreiben Sie jede Funktion so, dass der Code, der eine Exception auslösen könnte, erst einmal isoliert und in Sicherheit arbeitet. Erst dann, wenn Sie wissen, dass die eigentliche Arbeit erledigt wurde, sollten Sie den Programmzustand verändern (sowie Aufräumarbeiten durchführen), und zwar mit Hilfe von Operationen, die ihrerseits keine Exceptions auswerfen.

Lektion 10: Entwurf exception-sicheren Codes – Teil 3 Schwierigkeitsgrad: 9½

Na, blicken Sie langsam durch bei der Exception-Sicherheit? Dann ist es an der Zeit für eine richtig harte Nuss.

Jetzt kommt der letzte Teil von Cargills ursprünglichem `Stack`-Template an die Reihe.

```
template <class T> class Stack
{
public:
  Stack();
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Size() const;
  void   Push(const T&);
  T      Pop();   // wirft Exception wenn leer
private:
  T*     v_;      // Zeiger auf einen Speicherbereich
  size_t vsize_;  // groß genug für 'vsize_' T's
  size_t vused_;  // # der tatsächlich benutzten T's
};
```

Lösung

Size()

Die von allen Elementfunktionen von `Stack` am einfachsten zu implementierende Funktion ist `Size()`, da hier lediglich ein eingebauter Datentyp kopiert wird und keine Exceptions auftreten können.

```
template<class T>
size_t Stack<T>::Size() const
{
  return vused_;  // sicher, da eingebauter Datentyp
}
```

Push()

Bei `Push()` jedoch müssen wir unsere neu gewonnene Pflicht zur Sorgfalt erfüllen.

```
template<class T>
void Stack<T>::Push( const T& t )
{
  if( vused_ == vsize_ )  // wenn nötig um einen
  {                       // Faktor wachsen
    size_t vsize_new = vsize_*2+1;
    T* v_new = NewCopy( v_, vsize_, vsize_new );
    delete[] v_;  // kein throw
    v_ = v_new;   // Zuständigkeit übernehmen
    vsize_ = vsize_new;
  }
  v_[vused_] = t;
  ++vused_;
}
```

Wenn kein weiterer freier Speicherplatz mehr zur Verfügung steht, berechnen wir zunächst eine neue Puffergröße und fertigen dann mit `NewCopy` eine größere Kopie an. Wenn `NewCopy` eine Exception auswirft, verbleibt der Zustand unseres `Stack`-Objekts wieder unverändert und die Exception wird sauber weitergeleitet. Das Löschen des ursprünglichen Puffers und die Übernahme des neuen erfordert nur Operationen, die bekanntermaßen keine Exceptions auslösen, so dass der gesamte `if`-Block exceptionsicher ist.

Im Anschluss an den Vergrößerungscode versuchen wir den neuen Wert zu kopieren und inkrementieren erst hinterher den Zähler `vused_`. Auf diese Weise wird sichergestellt, dass bei einer während der Zuweisung auftretenden Exception die Inkrementierung nicht durchgeführt und damit der Objektzustand nicht verändert wird. Verläuft dagegen die Zuweisung erfolgreich, wird der Zustand des `Stack`s angepasst, um die Anwesenheit des neuen Wertes zu vermerken. Anschließend ist dann alles in bester Ordnung.

☑ Richtlinie

Beachten Sie die kanonischen Exception-Sicherheitsregeln: Schreiben Sie jede Funktion so, dass der Code, der eine Exception auslösen könnte, erst einmal isoliert und in Sicherheit arbeitet. Erst dann, wenn Sie wissen, dass die eigentliche Arbeit erledigt wurde, sollten Sie den Programmzustand verändern (sowie Aufräumarbeiten durchführen), und zwar mit Hilfe von Operationen, die ihrerseits keine Exceptions auswerfen.

Pop() tanzt aus der Reihe

Eine Funktion ist noch übrig. Das war ja bis jetzt nicht so schwer, oder? Aber freuen Sie sich nicht zu sehr, denn wie sich herausstellt, ist `Pop()` in Hinsicht auf die Exception-Sicherheit die problematischste dieser Funktionen. Unser erster Ansatz könnte etwa so aussehen:

```
// Hmmm... wie sicher ist das wirklich?
template<class T>
T Stack<T>::Pop()
{
  if( vused_ == 0)
  {
    throw "pop bei leerem stack";
  }
  else
  {
    T result = v_[vused_-1];
    --vused_;
    return result;
  }
}
```

Ist der Stack leer, werfen wir eine entsprechende Exception. Ansonsten erzeugen wir eine Kopie der zurückzugebenden `T`-Instanz, aktualisieren den Zustand und geben die Instanz zurück. Falls diese erste Kopie von `v_[vused_-1]` fehlschlägt, wird die zugehörige Exception weitergeleitet und der Objektzustand unverändert belassen, was genau dem entspricht, was wir wollen.

Das funktioniert also, nicht wahr? Nun, fast. Es gibt einen subtilen Fehler, der vollständig außerhalb des Zuständigkeitsbereiches von `Stack::Pop()` liegt. Betrachten Sie den folgenden Code:

```
string s1(s.Pop());
string s2;
s2 = s.Pop();
```

Oben war nicht ohne Grund von der »ersten Kopie« (von `v_[vused_-1]`) die Rede, denn es gibt in beiden eben gezeigten Fällen jeweils noch eine zweite Kopie[5], nämlich die Kopie des zurückgegebenen temporären Objekts in das Zielobjekt. Wenn diese Copy-Konstruktion bzw. Copy-Zuweisung fehlschlägt, ist der Zustand des `Stack`s bereits aktualisiert, d.h. das oberste Element bereits entfernt, jedoch geht dieses oberste Element verloren, da es nie sein Ziel erreicht. Das sind schlechte Neuigkeiten, denn es bedeutet, dass jede Version von `Pop()`, die so wie diese ein temporäres Objekt zurückgibt und somit für zwei Seiteneffekte verantwortlich ist, nicht vollständig exception-sicher gemacht werden kann. Denn selbst wenn die Implementierung der Funktion selbst exception-sicher aussieht, wird trotzdem der Anwender dazu gezwungen, exception-unsicheren Code zu schreiben. Verallgemeinert gesagt sollten verändernde Funktionen `T`-Objekte nicht per Wert zurückgeben (Lektion 19 befasst sich näher mit Aspekten der Exception-Sicherheit bei Funktionen mit mehrfachen Seiteneffekten).

Daraus ergibt sich eine überaus bedeutende Schlussfolgerung: *Exception-Sicherheit beeinflusst das Klassen-Design*. Mit anderen Worten, Sie müssen die Exception-Sicherheit von Anfang an in den Entwurf mit einbeziehen, sie ist keinesfalls »lediglich ein Implementierungsdetail«.

☒ Typischer Fehler

Exception-Sicherheit darf Ihnen nicht erst hinterher einfallen. Exception-Sicherheit beeeinflusst bereits den Entwurf einer Klasse und ist nicht »nur ein Implementierungsdetail«.

Das eigentliche Problem

Eine Alternative – und gleichzeitig die minimalste Änderung[6] – besteht darin, die Spezifikation von `Pop()` wie folgt zu ändern:

```
template<class T>
void Stack<T>::Pop( T& result )
{
  if( vused_ == 0)
  {
```

5. Der erfahrene Leser wird natürlich korrekt bemerken, dass es sich tatsächlich um "null oder eine Kopie« handelt, da es dem Compiler freisteht, die zweite Kopie wegzuoptimieren. Entscheidend dabei ist, dass es eine Kopie geben *kann*, die man nicht ignorieren darf.
6. Eigentlich die minimale akzeptierbare Änderung. Sie könnten ja auch einfach die ursprüngliche Version so abändern, dass `T&` statt `T` zurückgegeben wird (das wäre dann eine Referenz auf das entfernte `T`-Objekt, das zu der Zeit physikalisch immer noch intern vorhanden ist), so dass der aufrufende Code exception-sicher ist. Aber das Zurückgeben von Referenzen auf eigentlich nicht mehr vorhandene Ressourcen ist eine böse Sache. Wenn sich die Implementierung später verändert, ist das vielleicht nicht mehr möglich! Hüten Sie sich davor.

```
    throw "pop bei leerem stack";
  }
  else
  {
    result = v_[vused_-1];
    --vused_;
  }
}
```

Dadurch wird sichergestellt, dass sich der Zustand des `Stack`s nicht ändert, bevor die Kopie sicher beim Aufrufer angekommen ist.

Aber das eigentliche Problem besteht darin, dass `Pop()` in seiner derzeitigen Spezifikation *zwei* Verantwortungen zu tragen hat, es muss nämlich das oberste Element entfernen und dessen Wert zurückgeben.

☑ Richtlinie

Setzen Sie auf Kohäsion. Bemühen Sie sich immer, jedem Codeteil – jedem Modul, jeder Klasse und jeder Funktion – eine einzige, wohl definierte Aufgabe zu geben.

Alternativ kann man daher (und meiner Meinung nach ist das die zu bevorzugende Methode) die Abfrage und das Entfernen des obersten Wertes auf zwei separate Funktionen aufteilen.

```
template<class T>
T& Stack<T>::Top()
{
  if( vused_ == 0)
  {
    throw "leerer stack";
  }
  return v_[vused_-1];
}

template<class T>
void Stack<T>::Pop()
{
  if( vused_ == 0)
  {
    throw "pop bei leerem stack";
  }
  else
  {
    --vused_;
  }
}
```

Haben Sie sich übrigens jemals darüber geärgert, dass die `Pop()`-Funktionen der Container der Standardbibliothek (zum Beispiel `list::pop_back`, `stack::pop` usw.) den ent-

fernten Wert nicht zurückgeben? Nun, einer der Gründe dafür ist, dass man die Exception-Sicherheit nicht schwächen wollte.

Vielleicht haben Sie auch schon gemerkt, dass die oben vorgestellten separaten Funktionen die gleichen Signaturen aufweisen wie die `top()`- und `pop()`-Elementfunktionen des `stack<>`-Adapters der Standardbibliothek. Das ist kein Zufall. Tatsächlich sind wir nur noch zwei öffentliche Elementfunktionen von der vollständigen öffentlichen Schnittstelle des `stack<>`-Adapters entfernt. Es fehlt noch

```
template<class T>
const T& Stack<T>::Top() const
{
  if( vused_ == 0)
  {
    throw "leerer stack";
  }
  else
  {
    return v_[vused_-1];
  }
}
```

um `Top()` auch für konstante `Stack`-Objekte durchführen zu können, sowie:

```
template<class T>
bool Stack<T>::Empty() const
{
  return( vused_ == 0 );
}
```

Natürlich ist die `stack<>`-Klasse des Standards in Wirklichkeit ein Container-Adapter, der einen anderen Container zur Implementierung benutzt. Aber die öffentliche Schnittstelle ist die gleiche und der Rest ist ja dann nur ein Implementierungsdetail.

Ich möchte noch auf einen weiteren sehr wichtigen Aspekt aufmerksam machen und gebe Ihnen das Folgende zum Nachdenken:

☒ Typischer Fehler

»Exception-Unsicherheit« und »schlechtes Design« gehen Hand in Hand. Ist ein Codeteil nicht exception-sicher, stellt das keinen Fehler dar, der nicht schnell behoben werden könnte. Wenn ein Codeteil aber nicht exception-sicher gemacht werden kann, weil es das zugrunde liegende Design nicht zulässt, ist das fast immer ein Zeichen für schlechtes Design. Beispiel 1: Eine Funktion, die zwei Verantwortungen zu tragen hat, kann nur schwer exception-sicher gemacht werden. Beispiel 2: Ein Copy-Zuweisungsoperator, der sich gegen Selbstzuweisung absichern muss, kann nicht exception-sicher sein.

Beispiel 2 wird sehr bald in dieser Miniserie demonstriert werden. Copy-Zuweisungsoperatoren können zwar sehr wohl auf Selbstzuweisung prüfen, auch wenn es gar nicht notwendig ist – zum Beispiel aus Effizienzgründen. Aber ein Copy-Konstruktor, der auf Selbstzuweisung prüfen *muss* (weil er ansonsten bei Selbstzuweisung nicht korrekt funktioniert), kann nicht exception-sicher sein.

Lektion 11: Entwurf exception-sicheren Codes – Teil 4 Schwierigkeitsgrad: 8

Kurze Unterbrechung in der Mitte der Serie: Was haben wir bisher erreicht?

Beantworten Sie nun, da wir einen exception-sicheren und exception-neutralen `Stack<T>`-Container implementiert haben, die folgenden Fragen so genau wie möglich:

1. Wie lauten die wichtigen Exception-Sicherheits-Garantien?
2. Welche Anforderungen sind für den eben implementierten Container `Stack<T>` an den enthaltenen Typ `T` zu stellen?

Lösung

So wie es mehrere Möglichkeiten gibt, einer Katze das Fell abzuziehen (ich habe das Gefühl, demnächst E-Mails wütender Tierschützer zu bekommen), so gibt es auch mehrere Wege, exception-sicheren Code zu schreiben. Wenn Exception-Sicherheit garantiert werden muss, dann existieren zwei Alternativen, aus denen wir wählen können. Zuerst wurden diese Garantien in dieser Form von Dave Abrahams dargestellt.

1. *Grundlegende Garantie: Auch in der Gegenwart von von T ausgeworfenen Exceptions oder von anderen Exceptions verursachen Stack-Objekte keine Ressourcenlecks.* Das hat zur Folge, dass der Container auch dann noch benutzbar und dekonstruierbar ist, wenn eine Exception während einer Containeroperation auftritt. Nach dem Auswurf einer Exception befindet sich der Container damit in einem konsistenten, aber nicht notwendigerweise vorhersagbaren Zustand. Container mit grundlegender Garantie können in einigen Umgebungen sicher funktionieren.
2. *Hohe Garantie: Tritt während einer Operation eine Exception auf, bleibt der Zustand des Programms unverändert.* Das impliziert immer auch eine Vorgehensweise des »Anvertrauens oder Zurücknehmens« sowie weiterhin, dass Referenzen oder Iteratoren auf einen Container gültig bleiben, wenn eine Operation fehlschlägt. Wenn zum Beispiel ein `Stack`-Anwender erst `Top()` und dann `Push()` aufruft, Letzteres aber wegen einer Exception scheitert, dann muss der Zustand des `Stack`-Objektes unverändert und die vom vorherigen Aufruf von `Pop()` stammende Referenz gültig bleiben. Weitere Informationen zu diesen Garantien enthält Dave Abrahams Doku-

mentation der exception-sicheren Version der Standardbibliothek von SGI unter `http://www.stlport.org/doc/exception_safety.html`.

Die hier vielleicht interessanteste Tatsache ist die, dass die Implementierung der grundlegenden Garantie oft die Einhaltung der hohen Garantie automatisch mit sich bringt.[7] Bei unserer `Stack`-Implementierung zum Beispiel war fast alles, was wir unternommen haben, nötig, um die grundlegende Garantie zu erfüllen – der gezeigte Code erfüllt aber bereits jetzt fast die hohe Garantie, dazu ist nur noch wenig bis gar keine zusätzliche Arbeit vonnöten.[8] Gar nicht übel, wenn bedenkt, wie viel Mühe uns das gekostet hat.

Zusätzlich zu diesen beiden Garantien gibt es noch eine dritte, der bestimmte Funktionen entsprechen müssen, damit Exception-Sicherheit insgesamt überhaupt möglich wird:

3. *Absolute Garantie*[9]*: Die Funktion wirft unter allen Umständen keine Exception aus.* Exception-Sicherheit ist insgesamt nicht möglich ohne Funktionen, die garantiert niemals Exceptions auswerfen. Wie wir gesehen haben, trifft das insbesondere für Destruktoren zu. Später wird sich in dieser Miniserie noch zeigen, dass sich auch einige Hilfsfunktionen wie zum Beispiel `Swap()` so verhalten müssen.

☑ Richtlinie

Verstehen Sie die grundlegende, die hohe und die absolute Exception-Sicherheits-Garantie.

Jetzt haben wir etwas, über das wir nachdenken können. Immerhin waren wir in der Lage, `Stack` nicht nur exception-sicher, sondern auch exception-neutral zu implementieren, und dabei haben wir nur einen `try/catch`-Block benutzt. Wie wir gleich sehen werden, können wir durch bessere Kapselung das `try` auch noch loswerden. Das bedeutet, wir können einen hochgradig exception-sicheren und exception-neutralen Container schreiben, ohne `try` oder `catch` zu verwenden – sehr schick, sehr elegant.

7. Beachten Sie bitte, dass ich »oft« meine, und nicht »immer«. Die Standardbibliothek liefert mit `vector` ein bekanntes Gegenbeispiel, bei dem die grundlegende Garantie nicht automatisch zur Einhaltung der hohen Garantie führt.
8. Es gibt da noch eine subtile Situation, in der diese `Stack`-Version nicht den Anforderungen der hohen Garantie entspricht. Wenn während des Ablaufs von `Push()` der interne Pufferspeicher vergrößert werden muss, die anschließende Zuweisung `v_[vused_] = t;` jedoch an einer Exception scheitert, befindet sich das entsprechende `Stack`-Objekt zwar in einem konsistenten Zustand, aber der interne Pufferspeicher wurde bewegt – was jede der von früheren `Top()`-Aufrufen stammenden Referenzen ungültig macht. Dieser letzte Fehler in `Stack::Push()` kann leicht durch Umsetzen von Code und Hinzufügen eines `try`-Blocks behoben werden. In der zweiten Hälfte dieser Miniserie wird jedoch eine bessere Lösung vorgestellt, die dieses Problem nicht hat und die Ansprüche der hohen Garantie erfüllt.
9. Sehr frei aus dem engl. Original “nothrow guarantee« übersetzt (Anm. d. Übers.).

Unser bisheriges `Stack`-Template fordert von seinem Instanziierungstyp `T` das Folgende:

- Default-Konstruktor (um die `v_`-Puffer zu konstruieren)
- Copy-Konstruktor (wenn `Pop()` per Wert zurückgibt)
- Destruktor, der keine Exceptions wirft (um Exception-Sicherheit garantieren zu können)
- Exception-sichere Copy-Zuweisung (um die Werte in `v_` zu setzen; und sollte die Copy-Zuweisung eine Exception auswerfen, muss garantiert sein, dass der Zustand des Zielobjekts unverändert bleibt. Das ist die einzige Elementfunktion von `T`, die exception-sicher sein muss, damit `Stack` exception-sicher sein kann.)

In der zweiten Hälfte dieser Miniserie werden wir ebenfalls sehen, wie auch diese Anforderungen noch reduziert werden können, ohne die Exception-Sicherheit zu gefährden. Außerdem werfen wir einen eingehenderen Blick auf die Standardoperationen der Anweisung `delete[] x;`.

Lektion 12: Entwurf exception-sicheren Codes – Teil 5 Schwierigkeitsgrad: 7

In Ordnung, Sie haben sich genug ausgeruht – krempeln Sie die Ärmel hoch und machen Sie sich auf einen wilden Ritt gefasst.

Wir sind jetzt bereit, uns weiter in das gleiche Beispiel zu vertiefen und nicht nur eine, sondern zwei neue und verbesserte Versionen des `Stack`s zu schreiben. Tatsächlich ist es nicht einfach nur möglich, exception-sichere generische Container zu schreiben, sondern wir werden am Ende dieser Miniserie nicht weniger als drei vollständige Lösungen für dieses `Stack`-Problem gefunden haben.

Außerdem werden wir uns mit mehreren interessanten Fragestellungen beschäftigen:

- Wie können wir durch fortschrittlichere Techniken die Ressourcenverwaltung verbessern und obendrein die letzten `try/catch`-Blöcke loswerden?
- Wie lässt sich `Stack` verbessern, indem die Anforderungen an `T`, den enthaltenen Typ, reduziert werden?
- Sollten generische Container Exception-Spezifikationen benutzen?
- Was machen `new[]` und `delete[]` wirklich?

Die Antwort auf die letzte Frage unterscheidet sich wahrscheinlich von dem, was man zuerst erwarten würde. Das Schreiben exception-sicherer Container in C++ ist keine Raketenforschung, es erfordert lediglich große Sorgfalt und ein Verständnis für die Funktionsweise der Sprache. Dabei ist es hilfreich, es sich zur Angewohnheit zu machen, jeden Konstrukt mit einem gewissen Argwohn daraufhin zu betrachten, ob er

einen Funktionsaufruf darstellt. Hierbei ist auch auf die subtileren Schuldigen zu achten, zum Beispiel benutzerdefinierte Operatoren, benutzerdefinierte Umwandlungen und unauffällige temporäre Objekte, denn jeder Funktionsaufruf kann eine Exception auslösen.[10]

Eine Methode, einen exception-sicheren Container wie `Stack` stark zu vereinfachen, besteht in der Verwendung einer besseren Kapselung. Insbesondere möchten wir die grundlegende Speicherverwaltung isolieren. Bei unserer ursprünglichen exception-sicheren `Stack`-Version bestand die meiste Arbeit darin, die elementaren Speicherallozierungen korrekt zum Funktionieren zu bringen. Was liegt also näher, als eine Hilfsklasse einzuführen, um dort diese Arbeit zu konzentrieren?

```
template <class T> class StackImpl
{
/*????*/:
  StackImpl(size_t size=0);
  ~StackImpl();
  void Swap(StackImpl& other) throw();
  T*     v_;      // Zeiger auf einen Speicherbereich
  size_t vsize_;  // groß genug für 'vsize_' T's
  size_t vused_;  // # der tatsächlich benutzten T's
private:
  // privat und undefiniert: kein Kopieren erlaubt
  StackImpl( const StackImpl& );
  StackImpl& operator=( const StackImpl& );
};
```

`StackImpl` besitzt alle Datenelemente des ursprünglichen `Stack`-Templates, so dass wir im Wesentlichen dessen Repräsentation vollständig nach `StackImpl` verlegt haben. `StackImpl` verfügt zusätzlich über eine Hilfsfunktion namens `Swap()`, die die Innereien einer `StackImpl`-Instanz mit denen einer anderen vertauscht.

Ihre Aufgaben lauten nun:

1. Implementieren Sie alle drei Elementfunktionen von `StackImpl`, allerdings nicht auf die herkömmliche Weise. Der `v_`-Puffer soll genauso viele konstruierte `T`-Objekte enthalten, wie es `T`-Objekte im Container gibt, nicht mehr und nicht weniger. Insbesondere dürfen unbenutzte Positionen im `v_`-Puffer keine konstruierten `T`-Objekte enthalten.
2. Beschreiben Sie den Verantwortungsbereich von `StackImpl`. Welchen Zweck hat diese Klasse?
3. Was sollte an Stelle von `/*????*/` stehen? Welchen Effekt hat das dann darauf, wie `StackImpl` benutzt wird? Seien Sie so konkret wie möglich.

10. Natürlich mit Ausnahme der Funktionen, deren Exception-Spezifikation `throw()` lautet, und bestimmter Funktionen der Standardbibliothek, die entsprechend dokumentiert sind.

Lösung

Wir werden uns nicht lange damit aufhalten, zu analysieren, warum die folgenden Funktionen vollständig exception-sicher (funktionieren korrekt bei Anwesenheit von Exceptions) und exception-neutral (leiten alle Exceptions an den Aufrufer weiter) sind, da die Gründe so ziemlich dieselben sind wie die, die wir detailliert in der ersten Hälfte der Miniserie diskutiert haben. Nehmen wir uns trotzdem ein paar Minuten Zeit, um die Lösungen zu untersuchen. Und beachten Sie auch die Bemerkungen.

Konstruktor

Der Konstruktor ist relativ unkompliziert. Wir benutzen `operator new()`, um den Puffer als Rohspeicher zu allozieren. (Würden wir einen `new`-Ausdruck wie `new T[size]` verwenden, würde der Puffer mit über den Default-Konstruktor konstruierten T-Instanzen initialisiert werden, was durch die Aufgabenstellung explizit verboten ist.)

```
template <class T>
StackImpl<T>::StackImpl( size_t size )
  : v_( static_cast<T*>
          ( size == 0
            ? 0
            : operator new(sizeof(T)*size) ) ),
    vsize_(size),
    vused_(0)
{
}
```

Destruktor

Von den drei Funktionen ist der Destruktor die am einfachsten zu implementierende. Erinnern wir uns wieder, was wir an früherer Stelle über `operator delete()` gelernt haben. (Zu den Details solcher Funktionen wie `destroy()` und `swap()`, die in den nächsten Codeausschnitten verwendet werden, siehe den Kasten »Einige Standardhilfsfunktionen«.)

```
template <class T>
StackImpl<T>::~StackImpl()
{
    destroy( v_, v_+vused_ ); // kein throw
    operator delete( v_ );
}
```

Einige Standardhilfsfunktionen

Die in dieser Lösung dargestellten Klassen `Stack` und `StackImpl` benutzen drei Hilfsfunktionen, die direkt aus der Standardbibliothek entnommen bzw. abgeleitet wurden: `construct()`, `destroy()` und `swap()`. Diese Funktionen sehen in vereinfachter Form so aus:

```
// construct() konstruiert ein neues Objekt an
// einer vorgegebenen Position mit einem Anfangswert
template <class T1, class T2>
void construct( T1* p, const T2& value )
{
  new (p) T1(value);
}
```

Die obige Form von `new` wird »placement `new`« genannt. Hierbei wird kein neuer Speicher für das neue Objekt alloziert, sondern der Speicher benutzt, auf den der Zeiger (hier `p`) zeigt. Im Allgemeinen sollte jedes Objekt, das auf diese Weise erzeugt wurde, nicht mit `delete`, sondern durch expliziten Aufruf seines Destruktors zerstört werden (wie in den folgenden zwei Funktionen).

```
// destroy() zerstört ein Objekt oder einen Bereich
// von Objekten
template <class T>
void destroy( T* p )
{
  p->~T();
}
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    destroy( first );
    ++first;
  }
}
// swap() vertauscht zwei Werte
template <class T>
void swap( T& a, T& b )
{
  T temp(a); a = b; b = temp;
}
```

Von diesen Funktionen ist `destroy(first, last)` die interessanteste. Wir werden in Kürze wieder darauf zurückkommen, denn hinter dieser Funktion steckt mehr, als es den Anschein hat!

Um mehr über diese Standardfunktionen herauszufinden, nehmen Sie sich einige Minuten Zeit und untersuchen Sie, wie die entsprechende Implementierung in der von Ihnen benutzten Standardbibliothek aussieht. Das ist eine lohnende und lehrreiche Übung.

Swap

`Swap()` ist eine einfache, aber sehr wichtige Funktion, die wesentlich dafür verantwortlich zeichnet, dass die gesamte `Stack`-Klasse und insbesondere ihr Zuweisungsoperator derartig elegant gestaltet werden können.

```
template <class T>
void StackImpl<T>::Swap(StackImpl& other) throw()
{
    swap( v_,     other.v_ );
    swap( vsize_, other.vsize_ );
    swap( vused_, other.vused_ );
}
```

Zur Verdeutlichung der Funktionsweise stellen wir uns wie in Abbildung 1 gezeigt zwei `StackImpl<T>`-Objekte `a` und `b` vor.

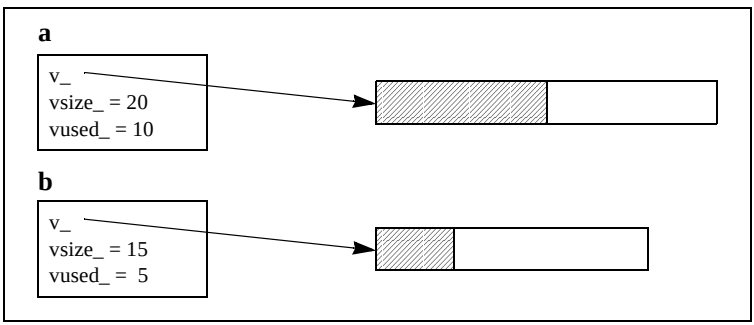

Abbildung 3.1: Zwei `StackImpl<T>`-Objekte `a` und `b`

Die Ausführung von `a.Swap(b)` führt dann zu dem Zustand, wie er in Abbildung 2 zu sehen ist.

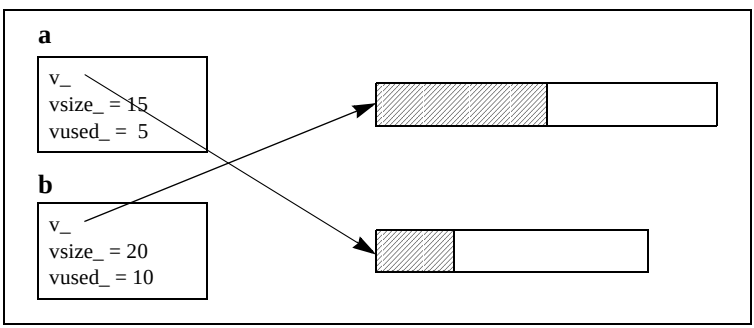

Abbildung 3.2: Die gleichen `StackImpl<T>`-Objekte `a` und `b` nach `a.Swap(b)`

`Swap()` bietet die stärkste aller Exception-Garantien, nämlich die absolute Garantie: Die Funktion wird niemals eine Exception auswerfen, egal unter welchen Umständen. Wie sich herausstellt, ist diese Eigenschaft für die `Stack`-Klasse unentbehrlich und stellt einen Grundpfeiler in deren Exception-Gebäude dar.

Was ist der eigentliche Zweck von `StackImpl`? Nun, es steckt nichts Geheimnisvolles dahinter: `StackImpl` ist für die Verwaltung des Rohspeichers und für die damit verbundenen Aufräumarbeiten verantwortlich. Jede anwendende Klasse braucht sich dadurch nicht mehr um diese Details zu kümmern.

☑ Richtlinie

Setzen Sie auf Kohäsion. Bemühen Sie sich immer, jedem Codeteil – jedem Modul, jeder Klasse und jeder Funktion – eine einzige, wohl definierte Aufgabe zu geben.

Welche Zugriffsspezifikation würden Sie also an die Stelle des Kommentars »`/*????*/`« setzen? Hinweis: Der Name `StackImpl` deutet auf eine gewisse »ist implementiert mit«-Beziehung hin und in C++ gibt es im Wesentlichen zwei Möglichkeiten, so etwas auszudrücken.

Methode 1: Private Basisklasse. An Stelle der fehlenden Zugriffsspezifikation muss entweder `protected` oder `public` stehen. (Bei `private` könnte niemand die Klasse benutzen.) Betrachten wir zuerst, was bei `protected` passiert.

`protected` bedeutet, dass `StackImpl` als private Basisklasse benutzt werden soll. `Stack` wird also »implementiert mit« `StackImpl`, genau das drückt die private Vererbung aus. Dabei entsteht eine klare Trennung der Verantwortlichkeiten. Die Basisklasse `StackImpl` kümmert sich um die Speicherverwaltung und während der Zerstörung des `Stack` auch um die Zerstörung aller verbleibenden `T`-Objekte. Die abgeleitete Klasse `Stack` sorgt dafür, dass innerhalb des Rohspeichers alle `T`-Objekte konstruiert werden. Die Rohspeicherverwaltung findet dadurch praktisch außerhalb von `Stack` statt, denn die anfängliche Allozierung muss zum Beispiel erst vollständig gelingen, bevor irgendein Konstruktor von `Stack` aufgerufen werden kann. Diese Version werden wir in Lektion 13 implementieren, die die letzte Phase dieser Miniserie einleitet.

Methode 2: privates Element. Betrachten wir nun, was passiert, wenn die in `StackImpl` fehlende Zugriffsspezifikation `public` lautet.

`public` deutet auf die Absicht hin, die Klasse `StackImpl` als Struktur zu verwenden, da ihre Datenelemente öffentlich zugänglich sind. `Stack` wird also wieder »implementiert mit« `StackImpl`, diesmal durch eine »hat ein«-Beziehung anstatt durch private Vererbung. Auch besteht hier die gleiche eindeutige Trennung der Verantwortlichkeiten. Das `StackImpl`-Objekt kümmert sich um die Speicherverwaltung und während der Zerstörung des `Stack`-Objekts auch um die Zerstörung aller verbleibenden `T`-Objekte. Das

beinhaltende `Stack`-Objekt sorgt dafür, dass innerhalb des Rohspeichers alle `T`-Objekte konstruiert werden. Da die Datenelemente vor Eintritt in einen Konstruktorrumpf initialisert werden, findet die Rohspeicherverwaltung ebenfalls außerhalb von `Stack` statt, denn die anfängliche Allozierung muss zum Beispiel erst vollständig gelingen, bevor irgendein Konstruktor von `Stack` aufgerufen werden kann.

Wie wir anhand des Codes noch sehen werden, unterscheiden sich die beiden Methoden nur geringfügig.

Lektion 13: Entwurf exception-sicheren Codes – Teil 6 Schwierigkeitsgrad: 9

Nun zu einem noch besseren `Stack`, mit geringeren Anforderungen an `T` – ganz abgesehen von einem sehr eleganten `operator=()`.

Nehmen Sie an, dass der Kommentar `/*????*/` in `StackImpl` für `protected` steht. Implementieren Sie alle Elementfunktionen der folgenden `Stack`-Version, die `StackImpl` als private Basisklasse haben soll.

```
template <class T>
class Stack : private StackImpl<T>
{
public:
  Stack(size_t size=0);
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void   Push(const T&);
  T&     Top();   // wirft Exception wenn leer
  void   Pop();   // wirft Exception wenn leer
};
```

Vergessen Sie nicht, wie immer alle Funktionen exception-sicher und exception-neutral zu machen. (Hinweis: Es gibt eine sehr elegante Methode, um einen vollständig sicheren `operator=()` zu implementieren. Sehen Sie es?)

Lösung

Der Default-Konstruktor

Mit der Methode der privaten Basisklasse sieht unsere `Stack`-Klasse etwa so aus (zur Abkürzung ist der Code inline dargestellt):

```
template <class T>
class Stack : private StackImpl<T>
{
public:
  Stack(size_t size=0)
    : StackImpl<T>(size)
  {
  }
```

Der Default-Konstruktor von `Stack` ruft einfach den Default-Konstruktor von `StackImpl` auf, der den Stack-Zustand auf leer setzt und optional eine Anfangsallozierung durchführt. Die einzige Operation, die eine Exception auswerfen könnte, ist das `new` im Konstruktor von `StackImpl`, was aber aus der Sicht der Exception-Sicherheit von `Stack` irrelevant ist. Wenn eine Exception auftritt, kommt der `Stack`-Konstruktor gar nicht zur Ausführung, so dass kein `Stack`-Objekt entsteht. Allozierungsprobleme in der Basisklasse beeinflussen also `Stack` nicht. (Siehe zusätzlich auch Lektion 8 über das Verlassen eines Konstruktors über eine Exception.)

Die ursprüngliche Konstruktorschnittstelle von `Stack` ist übrigens leicht angepasst worden, um eine »Anfangsempfehlung« für die Größe des zu allozierenden Speichers berücksichtigen zu können. Wir werden davon gleich Gebrauch machen, wenn wir uns mit der `Push()`-Funktion befassen.

☑ Richtlinie

Befolgen Sie die kanonischen Exception-Sicherheitsregeln: Wenden Sie immer das »Ressourcenerwerb ist Initialisierung«-Idiom an, um das Eigentum an der Ressource von deren Verwaltung zu trennen.

Der Destruktor

Hier begegnen wir der ersten Eleganz: Wir brauchen gar keinen `Stack`-Destruktor zu schreiben. Der Default-Destruktor ist völlig ausreichend, da er den `StackImpl`-Destruktor aufruft, der seinerseits alle konstruierten Objekte zerstört und sogar den Speicher freigibt. Elegant.

Der Copy-Konstruktor

Der Copy-Konstruktor von `Stack` ruft nicht den Copy-Konstruktor von `StackImpl` auf. (Siehe dazu die vorherige Lösung und die Diskussion um die Aufgabe von `construct()`.)

```
Stack(const Stack& other)
  : StackImpl<T>(other.vused_)
{
  while( vused_ < other.vused_ )
  {
    construct( v_+vused_, other.v_[vused_] );
    ++vused_;
  }
}
```

Die Copy-Konstruktion ist jetzt effizient und korrekt. Das Schlimmste, was passieren kann, ist, dass ein `T`-Konstruktor scheitert, wobei dann der `StackImpl`-Destruktor alle erfolgreich konstruierten Objekte zerstört und den Rohspeicher freigibt. Ein großer Vorteil der Ableitung von `StackImpl` besteht darin, dass wir beliebig viele weitere Konstruktoren hinzufügen können, ohne in jeden den Code für die Aufräumarbeiten integrieren zu müssen.

Elegante Copy-Zuweisung

Der folgende Code wendet eine unglaublich elegante und sehr schöne Methode an, um einen vollständig sicheren Copy-Zuweisungsoperator zu erzeugen. Und diese Technik ist sogar noch cooler, wenn Sie sie vorher noch nie gesehen haben.

```
Stack& operator=(const Stack& other)
{
  Stack temp(other);  // macht die ganze Arbeit
  Swap( temp );       // kann keine Exceptions auswerfen
  return *this;
}
```

Haben Sie's verstanden? Nehmen Sie sich Zeit, um vor dem Weiterlesen darüber nachzudenken. Diese Funktion ist die Verkörperung einer sehr wichtigen Richtlinie, der wir bereits begegnet sind.

☑ Richtlinie

Beachten Sie die kanonischen Exception-Sicherheitsregeln: Schreiben Sie jede Funktion so, dass der Code, der eine Exception auslösen könnte, erst einmal isoliert und in Sicherheit arbeitet. Erst dann, wenn Sie wissen, dass die eigentliche Arbeit erledigt wurde, sollten Sie den Programmzustand verändern (sowie Aufräumarbeiten durchführen), und zwar mit Hilfe von Operationen, die ihrerseits keine Exceptions auswerfen.

Wunderbar elegant, wenn auch ein wenig subtil. Wir konstruieren eine temporäre Kopie von `other` und tauschen durch `Swap()` die eigenen Inhalte gegen die von `temp` aus. Wenn dann der Gültigkeitsbereich von `temp` verlassen wird und es sich selbst zerstört,

beseitigt es dadurch automatisch unseren alten Containerinhalt, während wir im neuen Zustand verbleiben.

Wird `operator=()` auf diese Weise exception-sicher gemacht, erledigt sich damit auch gleichzeitig das Problem mit der Selbstzuweisung (zum Beispiel: `Stack s; s = s;`) ohne weiteres Zutun. (Weil Selbstzuweisung äußerst selten ist, habe ich den traditionellen Test `if( this != &other )` weggelassen, da dieser wieder seine eigenen kleinen Probleme mit sich bringt. Siehe Lektion 38 bezüglich aller Details.)

Da die eigentliche Arbeit während der Konstruktion von `temp` verrichtet wird, kann der Zustand unseres Objektes durch eventuell ausgeworfene Exceptions (entweder auf Grund von Speicherallozierung oder durch `T`-Copy-Konstruktion) nicht beeinflusst werden. Weiterhin können keine Speicherlecks oder andere Probleme durch das `temp`-Objekt entstehen, da der Copy-Konstruktor von `Stack` bereits in hohem Maße (hohe Garantie) exception-sicher ist. Ist all das erledigt, tauschen wir lediglich die interne Repräsentation unseres Objekts mit der von `temp` aus. Dabei können keine Exceptions auftreten, weil `Swap()` eine Exception-Spezifikation von `throw()` hat und sowieso nur eingebaute Datentypen kopiert werden.

Man beachte, um wie vieles eleganter dieser Code gegenüber der exception-sicheren Copy-Zuweisung aus Lektion 9 ist. Bei der Überprüfung der Exception-Sicherheit erfordert unsere neue Version außerdem viel weniger Sorgfalt.

Wenn Sie zu den Leuten gehören, die ihren Code knapp mögen, können Sie die kanonische Form des `operator=()` auch kompakter so schreiben:

```
Stack& operator=(Stack other)
{
  Swap( other );
  return *this;
}
```

Stack<T>::Count()

Ja, `Count()` ist nach wie vor die einfachste Elementfunktion.

```
size_t Count() const
{
  return vused_;
}
```

Stack<T>::Push()

`Push()` erfordert etwas mehr Aufmerksamkeit. Studieren Sie vor dem Weiterlesen erst den Code ein wenig.

```
void Push( const T& t )
{
  if( vused_ == vsize_ )  // wächst wenn nötig
  {
    Stack temp( vsize_*2+1 );
    while( temp.Count() < vused_ )
    {
      temp.Push( v_[temp.Count()] );
    }
    temp.Push( t );
    Swap( temp );
  }
  else
  {
    construct( v_+vused_, t );
    ++vused_;
  }
}
```

Betrachten Sie zuerst den einfachen `else`-Fall: Wenn für das neue Objekt bereits Platz vorhanden ist, versuchen wir es zu konstruieren. Verläuft die Konstruktion erfolgreich, aktualisieren wir unseren `vused_`-Zähler. Das ist sicher und unkompliziert.

Wenn andererseits für das neue Objekt kein Platz mehr vorrätig ist, stoßen wir eine Reallozierung an. In diesem Fall konstruieren wir einfach ein temporäres `Stack`-Objekt, fügen das neue Element an der Spitze ein und tauschen dann unsere ursprünglichen Daten mit dem temporären Objekt, so dass sie ordnungsgemäß freigegeben werden.

Ist das nun exception-sicher? Ja, denn:

- Scheitert die Konstruktion von `temp`, bleibt unser Zustand unverändert. Es entstehen auch keine Ressourcenlecks.
- Wenn während des Aufbaus von `temp` (einschließlich der Kopie des neuen Objektes durch dessen Copy-Konstruktor) eine Exception auftritt, wird `temp` trotzdem korrekt durch seinen Destruktor zerstört, der automatisch beim Verlassen des Gültigkeitsbereiches aufgerufen wird.
- In keinem Fall verändern wir den Objektzustand, bevor nicht die gesamte Arbeit erfolgreich abgeschlossen wurde.

Wir gewähren dadurch die hohe Garantie (Anvertrauen oder Zurücknehmen), denn `Swap()` wird nur dann ausgeführt, wenn die gesamte Reallozierungs- und Push-Operation Erfolg hat. Würden wir für diesen Container Iteratoren unterstützen, würden diese bei einer fehlgeschlagenen Einfügeoperation ihre Gültigkeit nicht verlieren (zum Beispiel auf Grund einer notwendigen internen Vergößerung).

Stack<T>::Top()

Diese Funktion hat sich nicht verändert.

```
T& Top()
{
  if( vused_ == 0)
  {
    throw "leerer Stack";
  }
  return v_[vused_-1];
}
```

Stack<T>::Pop()

Bis auf den Aufruf von `destroy()` trifft das auch auf `Pop()` zu.

```
  void Pop()
  {
    if( vused_ == 0 )
    {
      throw "pop bei leerem stack";
    }
    else
    {
      --vused_;
      destroy( v_+vused_ );
    }
  }
};
```

Insgesamt gesehen hat sich `Push()` vereinfacht, doch am deutlichsten ist es am Konstruktor und Destruktor von `Stack` zu sehen, welchen Vorteil die Verlagerung des Ressourcenbesitzes in eine separate Klasse mit sich bringt. Dank `StackImpl` können wir nun so viele zusätzliche Konstruktoren schreiben, wie wir wollen, ohne dass wir uns wie beim letzten Mal in jedem davon um die Aufräumarbeiten zu kümmern haben.

Vielleicht haben Sie auch bemerkt, dass sogar der einzelne `try/catch`-Block, der in der ersten Version dieser Klasse noch nötig war, nun verschwunden ist – wir haben also ohne ein einziges `try` einen vollständig exception-sicheren und exception-neutralen Container geschrieben!

Lektion 14: Entwurf exception-sicheren Codes – Teil 7 Schwierigkeitsgrad: 5

Nur eine geringfügig geänderte Variante – natürlich immer noch mit einem sehr schönen `operator=()`.

Der Kommentar `/*????*/` in `StackImpl` soll nun für `public` stehen. Implementieren Sie alle Elementfunktionen der folgenden `Stack`-Version, die mit `StackImpl` implementiert werden soll, indem sie eine `StackImpl`-Elementvariable benutzt.

```
template <class T>
class Stack
{
public:
  Stack(size_t size=0);
  ~Stack();
  Stack(const Stack&);
  Stack& operator=(const Stack&);
  size_t Count() const;
  void   Push(const T&);
  T&     Top();   // wirft exception wenn leer
  void   Pop();   // wirft exception wenn leer
private:
  StackImpl<T> impl_;  // private Implementierung
};
```

Vergessen Sie die Exception-Sicherheit nicht.

Lösung

Diese Implementierung von `Stack` unterscheidet sich nur wenig von der ersten. Count() gibt zum Beispiel nicht das geerbte `vused_`, sondern `impl_.vused_` zurück.

Hier der vollständige Quelltext:

```
template <class T>
class Stack
{
public:
  Stack(size_t size=0)
    : impl_(size)
  {
  }
  Stack(const Stack& other)
    : impl_(other.impl_.vused_)
  {
    while( impl_.vused_ < other.impl_.vused_ )
    {
      construct( impl_.v_+impl_.vused_,
                 other.impl_.v_[impl_.vused_] );
      ++impl_.vused_;
    }
  }
  Stack& operator=(const Stack& other)
  {
    Stack temp(other);
```

```
    impl_.Swap(temp.impl_); // kein throw
    return *this;
  }
  size_t Count() const
  {
    return impl_.vused_;
  }
  void Push( const T& t )
  {
    if( impl_.vused_ == impl_.vsize_ )
    {
      Stack temp( impl_.vsize_*2+1 );
      while( temp.Count() < impl_.vused_ )
      {
        temp.Push( impl_.v_[temp.Count()] );
      }
      temp.Push( t );
      impl_.Swap( temp.impl_ );
    }
    else
    {
      construct( impl_.v_+impl_.vused_, t );
      ++impl_.vused_;
    }
  }
  T& Top()
  {
    if( impl_.vused_ == 0 )
    {
      throw "leerer stack";
    }
    return impl_.v_[impl_.vused_-1];
  }
  void Pop()
  {
    if( impl_.vused_ == 0 )
    {
      throw "pop bei leerem stack";
    }
    else
    {
      --impl_.vused_;
      destroy( impl_.v_+impl_.vused_ );
    }
  }
private:
  StackImpl<T> impl_; // private Implementierung
};
```

Wow! Das sind jede Menge `impl_`s. Und das bringt uns auch gleich zur letzten Frage dieser Miniserie.

Lektion 15: Entwurf exception-sicheren Codes – Teil 8 Schwierigkeitsgrad: 9

Das war's – hier kommt der letzte Abschnitt dieser Miniserie. Der Schluss ist ein guter Zeitpunkt, um innezuhalten und nachzudenken, und das ist genau das, was wir in den letzten drei Problemstellungen tun werden.

1. Welche Methode ist besser – `StackImpl` als private Basisklasse oder als Elementvariable zu benutzen?
2. In welchem Maße sind die letzten beiden `Stack`-Versionen wiederverwendbar? Welche Anforderungen stellen sie an den enthaltenen Typ `T`? (Oder mit anderen Worten: Welche Arten von `T` akzeptierte unser letzter `Stack`? Je geringer die Anforderungen sind, desto eher wird `Stack` wiederverwendbar sein.)
3. Sollten die Elementfunktionen von `Stack` mit Exception-Spezifikationen ausgestattet sein?

Lösung

Lassen Sie uns die Fragen der Reihe nach beantworten.

1. Welche Methode ist besser – `StackImpl` als private Basisklasse oder als Elementvariable zu benutzen?

   Beide Methoden erzielen im Wesentlichen den gleichen Effekt und trennen die Speicherverwaltung von der Objekt-Konstruktion/Zerstörung.

   Ich halte mich bei der Wahl zwischen privater Vererbung und Verwendung als Elementvariable an die Faustregel, Vererbung nur dann einzusetzen, wenn es absolut notwendig ist. Durch beide Methoden wird ein »ist implementiert mit« ausgedrückt, jedoch erzwingt die Verwendung als Elementvariable eine bessere Trennung der Zuständigkeiten, da die benutzende Klasse als normaler Anwender nur Zugriff auf die öffentliche Schnittstelle der benutzten Klasse hat. Verwenden Sie Vererbung nur dann, wenn es unbedingt erforderlich ist, also wenn

   – Sie Zugriff auf die protected-Elemente der Klasse brauchen,
   – Sie eine virtuelle Funktion überschreiben wollen, oder
   – das Objekt vor anderen Basisunterobjekten konstruiert werden muss.[11]

11. In unserem Fall ist es zugegebenermaßen verführerisch, private Vererbung schon allein wegen der syntaktischen Bequemlichkeit zu verwenden, damit wir nicht so oft »`impl_.`« schreiben müssen.

2. In welchem Maße sind die letzten beiden `Stack`-Versionen wiederverwendbar? Welche Anforderungen stellen sie an den enthaltenen Typ `T`?

   Stellen Sie sich beim Schreiben einer Template-Klasse, insbesondere bei einer, die potenziell als generischer Container zu benutzen ist, immer die entscheidende Frage: Wie wiederverwendbar ist meine Klasse? Oder anderes ausgedrückt: Welche Pflichten habe ich den Anwendern der Klasse auferlegt, und begrenzen diese Pflichten fälschlicherweise die Möglichkeiten, die den Anwendern bei einem vernünftigen Einsatz der Klasse zur Verfügung stehen?

   Diese `Stack`-Templates unterscheiden sich hauptsächlich in zwei Dingen von unserer früheren Version. Einen Unterschied haben wir bereits besprochen: die Trennung der Speicherverwaltung von der Erzeugung und Zerstörung der enthaltenen Objekte. Das ist zwar schön, interessiert aber den Anwender nicht wirklich. Der zweite Unterschied betrifft die Objektkonstruktion. Die beiden neuen `Stack`-Klassen konstruieren und zerstören die Objekte einzeln an Ort und Stelle, wenn sie gebraucht werden, anstatt im gesamten `T`-Pufferspeicher Default-`T`-Objekte zu erzeugen und sie bei Bedarf zuzuweisen.

   Daraus ergeben sich als entscheidende Vorteile eine höhere Effizienz und verringerte Anforderungen an den enthaltenen Typ `T`. Erinnern Sie sich, dass unser ursprünglicher `Stack` von `T` verlangte, vier Operationen zur Verfügung zu stellen:

   – Default-Konstruktor (zur Konstruktion der `v_`-Puffer)

   – Copy-Konstruktor (falls `Pop()` per Wert zurückgibt)

   – Destruktor, der keine Exceptions auswirft (um Exception-Sicherheit garantieren zu können)

   – Exception-sichere Copy-Zuweisung (um die Werte in `v_` zu setzen. Wenn die Copy-Zuweisung eine Exception auswirft, muss das Zielobjekt unverändert bleiben. Dies ist die einzige Elementfunktion von `T`, die zur Wahrung der Exception-Sicherheit unseres `Stack`s ebenfalls exception-sicher sein muss.)

   Ein Default-Konstruktor wird jetzt nicht mehr benötigt, da die einzige `T`-Konstruktion, die jemals ausgeführt wird, eine Copy-Konstruktion ist. Weiterhin ist nun keine Copy-Zuweisung mehr nötig, da `T`-Objekte innerhalb von `Stack` und `StackImpl` nicht mehr zugewiesen werden. Andererseits ist nun stets ein Copy-Konstruktor erforderlich. Unsere neuen `Stack`-Klassen stellen also letztlich die beiden folgenden Anforderungen an `T`:

   – Copy-Konstruktor

   – Destruktor, der keine Exceptions auswirft (zur Gewährleistung der Exception-Sicherheit)

Inwieweit erfüllt dies unsere ursprüngliche Forderung nach universeller Wiederverwendbarkeit? Nun, viele Klassen verfügen sowohl über einen Default-Konstruktor als auch über einen Copy-Zuweisungsoperator, bei vielen anderen nützlichen Klassen ist das aber nicht der Fall. (Tatsächlich gibt es Objekte, die man nicht zuweisen kann, da sie zum Beispiel als Elemente Referenzen enthalten, die nicht umgesetzt werden können.) Jetzt können auch solche Objekte in `Stack`s gespeichert werden, die ursprüngliche Version war dazu nicht geeignet. Das stellt definitiv einen großen Vorteil dar, den ziemlich viele Anwender zu schätzen wissen werden, wenn `Stack` mit der Zeit wiederverwendet wird.

☑ Richtlinie

Behalten Sie beim Entwurf auch die Wiederverwendbarkeit im Auge.

3. Sollten die Elementfunktionen von `Stack` mit Exception-Spezifikationen ausgestattet sein?

   Kurz gesagt: Nein, denn wir, die Autoren von `Stack`, haben nicht genug Informationen. Aber auch wenn wir die hätten, würden wir es wahrscheinlich nicht wollen. Das gilt im Prinzip für jeden generischen Container.

   Betrachten Sie zunächst, was wir, die Autoren von `Stack`, über den enthaltenen Typ `T` wissen: im Grunde herzlich wenig. Insbesondere wissen wir nicht genau, welche Operationen von `T` welche Exceptions werfen. Wir könnten natürlich ein wenig totalitär werden und weitere Vorschriften über `T` erlassen. Dadurch erhielten wir mehr Informationen über `T` und könnten die Elementfunktionen von `Stack` mit einigen nützlichen Exceptionspezifikationen ausstatten. Allerdings würde das unser Vorhaben sabotieren, `Stack` universell nutzbar zu machen, und kommt daher nicht in Frage.

   Wie Sie sicher bemerkt haben, geben einige Container-Funktionen (zum Beispiel `Count()`) ein Skalar zurück und werfen garantiert keine Exceptions aus. Kann man diese denn nicht als `throw()` deklarieren? Ja, aber aus zwei guten Gründen möchten Sie das wahrscheinlich nicht tun.

   – Indem Sie `throw()` schreiben, begrenzen Sie Ihre zukünftigen Möglichkeiten. Sie könnten zum Beispiel die zugrunde liegende Implementierung derart ändern wollen, dass nun doch Exceptions ausgeworfen werden können. Das Entfernen der Exception-Spezifikation ist dann riskant, da vorhandene Anwendungen eventuell nicht mehr funktionieren (denn die neue Klassenversion bricht ein altes Versprechen). Die Klasse wird dadurch spröde und unflexibel, da ihr eine gewisse Renitenz gegen Veränderungen innewohnt. (`throw()` bei virtuellen Funktionen kann eine Klasse ebenfalls weniger erweiterungsfähig machen, da

es denjenigen stark einschränkt, der Ableitungen von dieser Klasse schreiben will. Es kann Sinn machen, aber eine solche Entscheidung sollte sorgfältig durchdacht werden.)

– Exception-Spezifikationen können unabhängig davon, ob eine Exception geworfen wird oder nicht, zu einem Peformance-Overhead führen. Obwohl viele Compiler diesen Effekt immer besser minimieren, ist es bei oft benutzten Operationen und universellen Containern eventuell besser, auf Exception-Spezifikationen zu verzichten, um diesen Overhead zu vermeiden.

Lektion 16: Entwurf exception-sicheren Codes – Teil 9 Schwierigkeitsgrad: 8

Wie gut verstehen Sie den harmlosen Ausdruck `delete[] p`? Welche Auswirkungen hat er auf die Exception-Sicherheit?

Jetzt kommt das Thema, auf das Sie schon die ganze Zeit gewartet haben: »Warum sind Destruktoren, die Exceptions werfen, böse?«

Betrachten Sie die Anweisung `delete[] p;`, bei der `p` auf ein gültiges Array zeigt, das durch `new[]` korrekt alloziert und initialisiert wurde.

– Was geschieht bei `delete[] p;` wirklich?

– Wie sicher ist es? Seien Sie so präzise wie möglich.

Lösung

Hierbei handelt es sich um ein Schlüsselthema: Was tut das harmlos aussehende `delete[] p;` denn nun wirklich? Und wie sicher ist es?

Destruktoren, die Exceptions werfen, und warum sie so böse sind

Erinnern wir uns zuerst an unsere Standardhilfsfunktion `destroy()` (siehe den begleitenden Kasten):

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    destroy( first ); // ruft den Destruktor von "*first"
    ++first;
  }
}
```

Diese Funktion war in unserem obigen Beispiel sicher, weil wir verlangt haben, dass der Destruktor keine Exceptions auswirft. Aber was wäre, wenn wir das dem Destruktor eines enthaltenen Objekts erlauben würden? Nehmen wir einmal an, `destroy()` wird ein Bereich über fünf Objekte übergeben. Löst der erste Destruktor eine Exception aus, wird `destroy()` in der derzeitigen Fassung beendet und die anderen vier Objekte werden nicht zerstört. Das ist offensichtlich nicht so gut.

»Aber«, mögen Sie jetzt einwenden, »muss es nicht einen Weg geben, die Funktion `destroy()` so umzuschreiben, dass sie auch bei Exceptions werfenden `T`-Destruktoren korrekt arbeitet?« Nun, das ist nicht ganz so klar, wie es den Anschein hat. Anfangen könnte man etwa so:

```
template <class FwdIter>
void destroy( FwdIter first, FwdIter last )
{
  while( first != last )
  {
    try
    {
      destroy( first );
    }
    catch(...)
    {
      /* Was kommt hier hin? */
    }
    ++first;
  }
}
```

Nur was soll bei »Was geschieht hier« passieren? Es gibt nur drei Möglichkeiten: der `catch`-Block wirft die Exception erneut aus, konvertiert sie und wirft eine andere aus, oder er wirft nichts aus und setzt die Schleife fort.

1. Wirft der `catch`-Block die Exception erneut aus, erfüllt die `destroy()`-Funktion zumindest die Neutralitätsbedingung, da alle Exceptions von `T` normal weitergegeben werden. Allerdings ist sie nicht sicher, da sie beim Auftreten von Exceptions das Entstehen von Ressourcenlecks zulässt. Denn da es für `destroy()` keine Möglichkeit gibt, die Zahl der noch nicht erfolgreich zerstörten Objekte irgendwie weiterzugeben, können diese Objekte nie korrekt zerstört und die damit verbundenen Ressourcen nicht freigegeben werden, was unvermeidlich zu Ressourcenlecks führt. Definitiv nicht gut.

2. Konvertiert der `catch`-Block die Exception und wirft eine andere aus, sind weder die Neutralitäts- noch die Sicherheitsanforderungen erfüllt. Damit fällt diese Variante flach.

3. Wenn der `catch`-Block seinerseits gar keine Exceptions auslöst, entspricht die Funktion `destroy()` auf jeden Fall der Forderung, dass bei Exceptions keine Ressourcenlecks auftreten dürfen.[12] Offensichtlich ist sie dann aber nicht exception-neutral, da die Exceptions von `T` niemals weitergeleitet, sondern ignoriert werden. (Jedenfalls aus der Sicht des Aufrufers, und das selbst dann, wenn der `catch`-Block irgendeine Art Logbuch führt.)

   Einige Leute schlagen an dieser Stelle vor, dass die Funktion die Exception auffangen und »zwischenspeichern« sollte, um anschließend mit den Aufräumarbeiten fortzufahren. Die gespeicherte Exception wird dann am Ende erneut ausgelöst. Aber das ist auch keine Lösung, denn man kann dabei nicht korrekt mit mehreren Exceptions umgehen (auch wenn man sie alle bis zum Ende speichert, so kann man doch nur eine weiterleiten, so dass die anderen still und heimlich verschwinden). Auch wenn Sie jetzt vielleicht nach Alternativen suchen, glauben Sie mir, es läuft immer darauf hinaus, Code wie diesen zu schreiben, denn immer gibt es einen Satz von Objekten, von denen alle zerstört werden müssen. Wenn den `T`-Destruktoren erlaubt wird, Exceptions zu werfen, führt das stets irgendwo zu (im günstigsten Fall) exception-unsicherem Code.

   Damit wären wir dann auch schon beim harmlos wirkenden `new[]` und `delete[]`.

   Der entscheidende Punkt bei diesen beiden Anweisungen ist, dass sie das gleiche grundsätzliche Problem wie unsere Funktion `destroy()` haben. Betrachten Sie zum Beispiel den folgenden Code:

   ```
   T* p = new T[10];
   delete[] p;
   ```

   Sieht doch nach normalem, harmlosem C++ aus, oder? Aber haben Sie sich jemals gefragt, was bei `new[]` und `delete[]` geschieht, wenn ein `T`-Destruktor eine Exception auswirft? Selbst wenn ja, so werden Sie trotzdem keine Antwort finden, denn es gibt einfach keine. Laut Standard führt in diesem Code der Auswurf einer Exception seitens eines `T`-Destruktors zu undefiniertem Verhalten. Das bedeutet, dass jeder Code, in dem Arrays von Objekten alloziert oder freigegeben werden, deren Destruktoren Exceptions werfen können, sich undefiniert verhalten kann. Dies bedarf der näheren Erläuterung.

   Betrachten wir zunächst, was passiert, wenn alle Konstruktionen erfolgreich verliefen und dann der fünfte `T`-Destruktor während der `delete[]`-Operation eine Exception auslöst. `delete[]` hat hier das gleiche `catch`-Problem, wie es oben schon bei `destroy()` auftrat. Die Anweisung kann die Exception nicht weiterleiten, da dann

12. Natürlich kann es immer noch zu einem Leck kommen, wenn ein T-Destruktor eine Exception in der Weise auswirft, dass dabei seine Ressourcen nicht vollständig freigegeben werden. Das ist dann aber nicht das Problem von `destroy()`, sondern bedeutet nur, dass T selbst nicht exception-sicher ist. Die Funktion `destroy()` ist nach wie vor frei von Lecks, da sie die Ressourcen, für die sie zuständig ist (nämlich die `T`-Objekte selbst) vollständig freigibt.

die verbleibenden `T`-Objekte nicht zerstört und außerdem unwiederbringlich verloren gehen würden. Sie kann sie aber auch nicht umwandeln oder absorbieren, da sie dann nicht exception-neutral wäre.

Untersuchen wir weiterhin, was passiert, wenn der fünfte Konstruktor eine Exception auslöst. Der Destruktor des vierten Objektes wird daraufhin aufgerufen, dann der des dritten usw., bis alle erfolgreich konstruierten `T`-Objekte wieder zerstört wurden. Anschließend wird der Speicher freigegeben. Was aber, wenn dieser Vorgang nicht so glatt abläuft? Was geschieht insbesondere dann, wenn, nachdem der fünfte Konstruktor eine Exception ausgeworfen hat, auch der Destruktor des vierten Objekts eine Exception auslöst? Selbst wenn man diese ignoriert, so kann immer noch der dritte Destruktor eine Exception werfen. Sie sehen also, wohin das führt.

Wenn Destruktoren Exceptions werfen dürfen, kann weder `new[]` noch `delete[]` exception-sicher und exception-neutral gemacht werden.

Die Schlussfolgerung ist einfach: *Schreiben Sie niemals Destruktoren, die Exceptions durchlassen*.[13] Bei einer Klasse mit solch einem Destruktor ist das Anlegen eines Arrays von Instanzen mit `new[]` bzw. das Löschen mit `delete[]` nicht sicher möglich. Alle Destruktoren sollten immer so implementiert sein, als hätten sie eine Exception-Spezifikation von `throw()`, so dass keine Exceptions von ihnen ausgehen können.

☑ Richtlinie

Beachten Sie die kanonischen Exception-Sicherheitsregeln: Gestatten Sie niemals einer Exception, einen Destruktor oder einen überladenen `operator delete()` bzw. `operator delete[]()` zu verlassen. Schreiben Sie jeden Destruktor und jede Deallozierungsfunktion so, als hätten sie eine Exception-Spezifikation »`throw()`«.

Zugegeben, manch einer könnte diese Entwicklung der Dinge als ein wenig ungünstig empfinden, denn einer der ursprünglichen Gründe für die Einführung von Exceptions war ja der, sowohl Konstruktoren als auch Destruktoren die Möglichkeit zu geben, Fehler zu melden (denn beide haben keine Rückgabewerte). Das ist nicht ganz richtig, da damit hauptsächlich Konstruktorfehler addressiert werden sollten (schließlich sollen Destruktoren zerstören, da besteht definitiv weniger

13. Seit der Londoner Konferenz im Juli 1997 enthält der Standardentwurf diese Pauschalaussage: »Keine einzige Destruktor-Operation der Standard-C++-Bibliothek löst Exceptions aus.« Es haben nicht nur alle Standardklassen dieses Verhalten, sondern es ist auch untersagt, einen Standardcontainer mit einem Typ zu instantiieren, dessen Destruktor Exceptions werfen darf. Die restlichen Garantien, die ich umreißen werde, haben während der darauffolgenden Konferenz Substanz bekommen (Morristown, N.J., November 1997. Das war auch die Konferenz, in der der vollständige Standard beschlossen wurde.)

Raum für Misserfolge). Nun, zumindest bei Fehlern in Konstruktoren lassen sich Exceptions hervorragend einsetzen, auch beim Aufbau von Arrays mit oder ohne `new[]`, da sie hier in vorhersehbarer Weise funktionieren.

Sichere Exceptions

Will man erfolgreich exception-sicheren Code für Container und andere Objekte schreiben, ist manchmal große zusätzliche Sorgfalt nötig. Aber fürchten Sie sich nicht übermäßig vor Exceptions. Wenden Sie die angegebenen Richtlinien an – also isolieren Sie die Ressourcenverwaltung, benutzen Sie das »Aktualisieren eines temporären Objekts, dann austauschen«-Idiom und schreiben Sie keine Klassen, deren Destruktoren Exceptions werfen dürfen – und Sie sind auf dem besten Weg zu sicherem, funktionierendem Code, der exception-sicher und exception-neutral ist. Der daraus resultierende Vorteil für Ihre Bibliothek und für deren Benutzer kann durchaus konkret und den ganzen Aufwand wert sein.

Der Einfachheit wegen (und hoffentlich zum späteren Nachschlagen) hier die Zusammenfassung der »kanonischen Exception-Sicherheit« für Sie:

☑ Richtlinie

Beachten Sie die kanonischen Exception-Sicherheitsregeln: (1) Gestatten Sie niemals einer Exception, einen Destruktor oder einen überladenen `operator delete()` bzw. `operator delete[]()` zu verlassen. Schreiben Sie jeden Destruktor und jede Deallozierungsfunktion so, als hätten sie eine Exception-Spezifikation »`throw()`«. (2) Wenden Sie immer das »Ressourcenerwerb ist Initialisierung«-Idiom, um das Eigentum an der Ressource von deren Verwaltung zu trennen. (3) Schreiben Sie jede Funktion so, dass der Code, der eine Exception auslösen könnte, erst einmal isoliert und in Sicherheit arbeitet. Erst dann, wenn Sie wissen, dass die eigentliche Arbeit erledigt wurde, sollten Sie den Programmzustand verändern (sowie Aufräumarbeiten durchführen), und zwar mit Hilfe von Operationen, die ihrerseits keine Exceptions auswerfen.

Lektion 17: Entwurf exception-sicheren Codes – Teil 10 Schwierigkeitsgrad: 9½

Der Schluss – endlich. Danke, dass Sie sich mit dieser Miniserie befasst haben. Ich hoffe, es hat Ihnen gefallen.

An diesem Punkt fühlen Sie sich möglicherweise etwas ausgesaugt und ziemlich mitgenommen. Das ist verständlich. Deshalb folgt hier noch eine letzte Frage als Abschiedsgeschenk, damit sich jeder an die im gleichen Maße (wenn nicht sogar mehr)

erschöpften Leute erinnert, die das alles von Anfang an selbst herausfinden mussten und sich dann abgestrampelt haben, um in letzter Minute vernünftige Exception-Sicherheitsgarantien in die Standardbibliothek hineinzubekommen. Es ist dies auch der richtige Zeitpunkt, um noch einmal Dave Abrahams, Greg Colvin, Matt Austern und all den anderen außergewöhnlichen (im Original »exceptional«, Anm. d. Übers.) Leuten öffentlich dafür zu danken, dass sie dabei mitgeholfen haben, die momentanen Sicherheitsgarantien in die Standardbibliothek einzuarbeiten. Und sie schafften das buchstäblich Tage bevor der Standard im November 1997 auf der ISO WG21 / ANSI J16-Konferenz in Morristown, N.J., USA eingefroren wurde.

Ist die C++-Standardbibliothek exception-sicher?

Begründen Sie Ihre Antwort.

Lösung

Exception-Sicherheit und die Standardbibliothek

Sind die Container der Standardbibliothek exception-sicher und exception-neutral? Kurz gesagt: Ja.[14]

Alle Iteratoren, die von Standardcontainern zurückgegeben werden, sind exception-sicher und können kopiert werden, ohne dass dadurch eine Exception ausgelöst würde.

Alle Standardcontainer müssen für alle Operationen die grundlegende Garantie implementieren: Sie können stets zerstört werden und befinden sich auch beim Auftreten von Exceptions zu jedem Zeitpunkt in einem konsistenten (wenn nicht sogar vorhersagbaren) Zustand.

Um dies zu ermöglichen, müssen bestimmte Funktionen die absolute Garantie einhalten (sie dürfen keine Exceptions werfen) – einschließlich `swap()` (dessen Bedeutung in einer der vorangegangenen Lektionen deutlich wurde), `allocator<T>::deallocate()` (dessen Bedeutung am Anfang dieser Miniserie bei der Diskussion um `operator delete()` illustriert wurde) und bestimmter Operationen der Template-Parametertypen selbst (speziell der Destruktor, dessen Bedeutung in Lektion 16 in der Diskussion »Destruktoren, die Exceptions werfen, und warum sie so böse sind« dargelegt wurde).

14. Es geht mir hier um den Teil der Standardbibliothek, zu dem die Container und Iteratoren gehören. Über andere Teile der Bibliothek, wie *iostreams* und *facets*, sagt die Spezifikation aus, dass sie zumindest die grundlegende Exception-Sicherheitsgarantie einhalten.

Für alle Standardcontainer gilt außerdem die hohe Garantie für alle Operationen (mit zwei Ausnahmen). Sie arbeiten alle nach dem Motto »Anvertrauen oder Zurücknehmen«, so dass eine Operation wie `insert` entweder vollständig erfolgreich verläuft oder andernfalls den Programmzustand nicht verändert. Dabei gilt zusätzlich, dass fehlgeschlagene Operationen nicht die Gültigkeit jedweder Iteratoren beeinflussen, die bereits auf den Container verweisen.

Von dieser Regel gibt es nur zwei Ausnahmen. Erstens ist bei allen Containern das Einfügen ganzer Bereiche (*»iterator range« insert*) niemals exception-sicher im Sinne der hohen Garantie. Zweitens gilt ausschließlich für `vector<T>` und `deque<T>`, dass Einfügen und Löschen (entweder einzelner Elemente oder ganzer Bereiche) im Sinne der hohen Garantie exception-sicher sind, solange der Copy-Konstruktor und Zuweisungsoperator von `T` keine Exceptions werfen. Beachten Sie die Konsequenzen dieser besonderen Einschränkungen. Denn neben anderen Dingen bedeutet dies leider, dass Einfügen und Löschen zum Beispiel bei einem `vector<string>` oder `vector<int>` nicht exception-sicher (im Sinne der hohen Garantie) sind.

Wozu diese Einschränkungen? Weil die Fähigkeit, jede Art von Operation rückgängig machen zu können, nicht ohne zusätzlichen Speicherplatz- und Laufzeit-Overhead realisiert werden kann. Und der Standard wollte diesen Overhead im Namen der Exception-Sicherheit nicht erzwingen. Alle anderen Containeroperationen können im Sinne der hohen Garantie ohne Overhead exception-sicher gemacht werden. Wenn Sie also jemals einen Bereich von Elementen in einen Container einfügen oder wenn der Copy-Konstruktor oder der Zuweisungsoperator von `T` Exceptions auslösen kann und Sie ein solches Element in ein `vector<T>` oder `deque<T>` einfügen oder daraus löschen, wird der entsprechende Container hinterher nicht notwendigerweise einen vorhersagbaren Inhalt haben und Iteratoren, die auf ihn verweisen, könnten ungültig geworden sein.

Was heißt das für Sie? Nun, wenn Sie eine Klasse schreiben, die ein Containerelement besitzt, und Sie führen Bereichseinfügungen durch, oder wenn Sie eine Klasse schreiben, die ein `vector<T>` oder `deque<T>` als Element hat, wobei der Copy-Konstruktor oder der Zuweisungsoperator von `T` Exceptions werfen kann, dann sind Sie selbst dafür verantwortlich, durch zusätzliche Anstrengungen dafür zu sorgen, dass Ihre Klasse bei einer Exception einen vorhersagbaren Zustand beibehält. Zum Glück gestalten sich diese »zusätzlichen Anstrengungen« ziemlich simpel. Wann immer Sie in den Container einfügen oder aus ihm löschen wollen, erzeugen Sie zuerst eine Kopie des Containers, führen die gewünschte Operation mit dieser Kopie durch und benutzen dann `swap()`, um auf diese neue Version überzuwechseln, nachdem Sie wissen, dass Kopieren und Ändern erfolgreich verliefen.

Lektion 18: Codekomplexität – Teil 1 Schwierigkeitsgrad: 9

Dieses Problem unterbreitet eine interessante Herausforderung: Wie viele Ausführungspfade können in einer einfachen dreizeiligen Funktion vorkommen? Die Antwort wird Sie bestimmt überraschen.

Wie viele Ausführungspfade könnten sich im folgenden Code verbergen?

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last() << " ist überbezahlt" << endl;
  }
  return e.First() + " " + e.Last();
}
```

Um ein wenig Struktur in die Angelegenheit zu bringen, sollten Sie mit den folgenden drei Annahmen beginnen und dann darauf aufbauend selbst weitermachen.

- Die unterschiedliche Reihenfolge bei der Auswertung von Funktionsparametern wird ignoriert, ebenso fehlgeschlagene Destruktoren.
- Aufgerufene Funktionen werden als atomar angesehen.
- Um als verschieden zu gelten, muss ein Ausführungspfad aus einer einzigartigen Sequenz von Funktionsaufrufen bestehen, die alle in der gleichen Weise ausgeführt werden.

Lösung

Denken wir zunächst über die Auswirkungen der gegebenen Annahmen nach:

1. Unterschiedliche Reihenfolge bei der Auswertung von Funktionsparametern wird ignoriert, genauso wie Exceptions, die von Destruktoren geworfen werden.[15] Zusatzfrage für die Unerschrockenen: Wie viele weitere Ausführungspfade entstehen, wenn Exceptions bei Destruktoren zugelassen werden?
2. Aufgerufene Funktionen werden als atomar angesehen. Der Aufruf »`e.Title()`« könnte zum Beispiel aus verschiedenen Gründen eine Exception auslösen (die Funktion könnte zum Beispiel selbst eine Exception werfen, sie könnte eine Exception nicht auffangen, die von einer von ihr aufgerufenen anderen Funktion stammt, oder sie könnte per Wert zurückgeben, wobei der Konstruktor des dabei entstehen-

15. Erlauben Sie niemals einer Exception, einen Destruktor zu verlassen. Ein solcher Code kann nicht vernünftig zum Funktionieren gebracht werden. Mehr über »Destruktoren, die Exceptions werfen, und warum sie so böse sind« in Lektion 16.

den temporären Objektes eine Exception werfen könnte). Relevant ist jedoch lediglich, ob die Operation »`e.Title()`« in einer Exception resultiert oder nicht.

3. Um als verschieden zu gelten, muss ein Ausführungspfad aus einer einzigartigen Sequenz von Funktionsaufrufen bestehen, die alle in der gleichen Weise ausgeführt werden.

   Wie viele mögliche Ausführungspfade gibt es hier also? Antwort: 23 (Und das in nur drei Zeilen!).

Sie fanden:	*Einschätzung:*
3	durchschnittlich
4-14	exception-bewusst
15-23	Guru

   Die 23 bestehen aus:

   – 3 nicht durch Exceptions verursachten Ausführungspfaden

   – 20 unsichtbaren durch Exceptions verursachten Ausführungspfaden

Mit *nicht durch Exceptions verursachten Ausführungspfaden* meine ich Pfade, die ausgeführt werden, wenn keine Exception ausgeworfen werden. Sie resultieren aus dem normalen C++-Programmfluss. Mit *durch Exceptions verursachten Ausführungspfaden* sind die Pfade gemeint, die als Ergebnis einer geworfenen oder weitergeleiteten Exception auftreten. Diese Ausführungspfade werde ich getrennt betrachten.

Nicht durch Exceptions verursachte Ausführungspfade

Für diese Ausführungspfade muss man die C/C++Regeln für die Auswertung von Ausdrücken kennen.

```
if(  e.Title() == "CEO" || e.Salary() > 100000 )
```

1. Wenn `e.Title() == "CEO"` `true` liefert, braucht der zweite Teil der Bedingung nicht ausgewertet zu werden (in diesem Fall würde also `e.Salary()` nicht aufgerufen werden). Das `cout` wird jedoch ausgeführt.

   Mit passenden Überladungen von `==`, `||` und/oder `>` in der Bedingung von `if` könnte sich das `||` als Funktionsaufruf herausstellen. In einem solchen Fall käme keine Abkürzung mehr zum Einsatz, sondern beide Teile der Bedingung würden jedesmal ausgewertet werden.

2. Wenn `e.Title != "CEO"` aber `e.Salary() > 100000`, würden beide Teile der Bedingung ausgewertet und das `cout` ausgeführt.

3. Wenn `e.Title != "CEO"` und `e.Salary() <= 100000`, wird das `cout` nicht ausgeführt.

Durch Exceptions verursachte Ausführungspfade

```
String EvaluateSalaryAndReturnName( Employee e )
  ^*^                                   ^4^
```

4. Das Argument wird per Wert übergeben, wodurch der Copy-Konstruktor von `Employee` zum Einsatz kommt. Diese Operation könnte eine Exception auslösen.

   `*`. Der Copy-Konstruktor von `String` könnte eine Exception werfen, während er den temporären Rückgabewert zum Aufrufer kopiert. Wir werden diesen einen Vorgang ignorieren, da er außerhalb dieser Funktion abläuft (und wie sich herausstellt, sind noch genügende Ausführungspfade übrig, für die wir zuständig sind).

```
if(  e.Title() == "CEO"  || e.Salary() > 100000 )
        ^5^   ^7^  ^6^  ^11^    ^8^  ^10^  ^9^
```

5. Die Elementfunktion `Title()` kann entweder selbst eine Exception werfen oder ein Klasseninstanz per Wert zurückgeben, deren Copy-Konstruktor eine Exception wirft.

6. Um Übereinstimmung mit einem gültigen `operator==()` zu erreichen, kann es erforderlich sein, die String-Konstante in eine temporäre Instanz einer Klasse (wahrscheinlich die gleiche wie der Rückgabetyp von `e.Title()`) umzuwandeln. Die Konstruktion dieses temporären Objektes könnte eine Exception auslösen.

7. Wenn `operator==()` eine vom Programmierer bereitgestellte Funktion ist, könnte sie eine Exception auswerfen.

8. In ähnlicher Weise wie in #5 kann `Salary()` entweder selbst eine Exception werfen oder eine Klasseninstanz per Wert zurückgeben, deren Copy-Konstruktor eine Exception wirft.

9. Ähnlich zu #6 könnte ein temporäres Objekt nötig werden, dessen Konstruktion eine Exception werfen könnte.

10. Dies könnte ähnlich zu #7 eine vom Programmierer zur Verfügung gestellte Funktion sein, die eine Exception werfen kann.

11. Ähnlich zu #7 und #10 könnte dies ebenfalls eine vom Programmierer zur Verfügung gestellte Funktion sein, die eine Exception werfen kann.

```
cout << e.First() << " " << e.Last() << " ist überbezahlt" << endl;
     ^12^ ^17^   ^13^   ^14^  ^18^  ^15^                  ^16^
```

12-16. Entsprechend dem C++-Standard könnte jeder der fünf Aufrufe von `operator<<` eine Exception auswerfen.

17-18. Ähnlich zu #5 könnten `First()` und/oder `Last()` eine Exception werfen oder jeweils ein Klasseninstanz per Wert zurückgeben, deren Copy-Konstruktor eine Exception wirft.

```
return e.First()   +   " "   +   e.Last();
         ^19^    ^22^ ^21^ ^23^   ^20^
```

19-20. Ähnlich zu #5 könnten `First()` und/oder `Last()` eine Exception werfen oder jeweils ein Klasseninstanz per Wert zurückgeben, deren Copy-Konstruktor eine Exception wirft.

21. Ähnlich zu #6 könnte ein temporäres Objekt nötig werden, dessen Konstruktion eine Exception werfen könnte.

22-23. Dies könnte ähnlich zu #7 eine vom Programmierer zur Verfügung gestellte Funktion sein, die eine Exception werfen kann.

☑ Richtlinie

Seien Sie stets auf Exceptions gefasst. Machen Sie sich bewusst, dass Code Exceptions auswerfen könnte.

Ein Ziel dieser Lektion war es, Ihnen zu demonstrieren, wie viele verschiedene Ausführungspfade schon in einfachem Code existieren können, wenn die Programmiersprache Exceptions unterstützt. Beeinflusst diese unsichtbare Komplexität die Zuverlässigkeit und Testbarkeit der Funktion? Die Antwort darauf liefert die nächste Lektion.

Lektion 19: Codekomplexität – Teil 2 Schwierigkeitsgrad: 7

Die Herausforderung: Machen Sie die Dreizeilenfunktion aus Lektion 18 exception-sicher. Bei dieser Übung werden einige wichtige Aspekte der Exception-Sicherheit veranschaulicht.

Ist Funktion aus Lektion 18 exception-sicher (korrekte Funktionsweise in Gegenwart von Exceptions) und exception-neutral (alle Exceptions werden an den Aufrufer weitergeleitet)?

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    cout << e.First() << " " << e.Last() << " ist überbezahlt" << endl;
```

```
    }
    return e.First() + " " + e.Last();
  }
```

Begründen Sie Ihre Antwort. Für den Fall, dass die Funktion exception-sicher ist, unterstützt sie dann die grundlegende, die hohe oder die absolute Garantie? Wenn nicht, wie müsste sie dann verändert werden, um einer dieser Garantien zu entsprechen?

Nehmen Sie dabei an, dass alle aufgerufenen Funktionen exception-sicher sind (sie können zwar Exceptions werfen, verursachen dann aber keine Seiteneffekte). Weiterhin sollen alle verwendeten Objekte, einschließlich der temporären, exception-sicher sein (sie geben ihre Ressourcen korrekt frei, wenn sie zerstört werden).

Siehe Lektion 11 zur Rekapitulation der grundlegenden, hohen und absoluten Garantie. Kurz gesagt stellt die grundlegende Garantie sicher, dass beim Zerstören keine Lecks entstehen, die hohe Garantie fügt dem eine vollständige Anvertrauen-oder-Zurücknehmen-Semantik hinzu und die absolute Garantie sichert zu, dass eine Funktion keine Exceptions auswirft.

Lösung

Zuerst einige Anmerkungen zu den Annahmen, bevor wir uns der eigentlichen Lösung zuwenden.

Entsprechend der Problemstellung werden wir annehmen, dass alle beteiligten Funktionen exception-sicher sind (sie können zwar Exceptions werfen, verursachen dann aber keine Seiteneffekte), ebenso alle verwendeten Objekte, einschließlich der temporären (sie geben ihre Ressourcen korrekt frei, wenn sie zerstört werden).

Streams behindern das Ganze gegebenenfalls, da sie nicht umkehrbare Seiteneffekte aufweisen können. Zum Beispiel könnte `operator<<` nach der Ausgabe eines Teils eines Strings eine Exception werfen und die bereits durchgeführte Ausgabe kann schließlich nicht rückgängig gemacht werden. Außerdem könnte der Stream danach in den Fehlerzustand versetzt worden sein. Wir werden diese Aspekte größtenteils vernachlässigen, denn diese Diskussion dreht sich hauptsächlich darum, wie man eine Funktion exception-sicher macht, die zwei verschiedene Seiteneffekte aufweist.

Hier also noch einmal die Frage: Ist Funktion aus Lektion 18 exception-sicher (korrekte Funktionsweise in Gegenwart von Exceptions) und exception-neutral (alle Exceptions werden an den Aufrufer weitergeleitet)?

```
String EvaluateSalaryAndReturnName( Employee e )
{
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
```

```
    cout << e.First() << " " << e.Last() << " ist überbezahlt" << endl;
  }
  return e.First() + " " + e.Last();
}
```

Im derzeitigen Zustand hält sich die Funktion an die grundlegende Garantie: Beim Auftreten von Exceptions kommt es nicht zu Ressourcenlecks.

Die Funktion entspricht allerdings *nicht* der hohen Garantie. Diese Garantie besagt, dass der Zustand des Programms nicht verändert werden darf, wenn die Funktion aufgrund einer Exception scheitert. `EvaluateSalaryAndReturnName()` hat jedoch, wie der Funktionsname schon vermuten lässt, zwei verschiedene Seiteneffekte.

- Eine Meldung »...`überbezahlt`...« wird nach `cout` ausgegeben.
- Ein Namens-String wird zurückgegeben.

Was den zweiten Seiteneffekt angeht, erfüllt die Funktion bereits die hohe Garantie, da der Wert beim Auftreten einer Exception niemals zurückgegeben wird. In Bezug auf den ersten Seiteneffekt ist die Funktion allerdings aus zwei Gründen nicht exceptionsicher:

- Die Meldung wird nur teilweise nach `cout` ausgegeben, wenn eine Exception ausgeworfen wird, nachdem der erste Teil der Meldung nach `cout` ausgegeben wurde, jedoch bevor die Meldung vollständig ausgegeben werden konnte (wenn zum Beispiel der fünfte Aufruf von `operator<<` eine Exception wirft).[16]
- Wenn die Meldung zwar erfolgreich ausgegeben werden konnte, jedoch später in der Funktion eine Exception ausgeworfen wird (zum Beispiel während der Montage des Rückgabewertes), dann wurde eine Meldung nach `cout` ausgegeben, obwohl die Funktion aufgrund einer Exception fehlschlug.

Die absolute Garantie erfüllt die Funktion überhaupt nicht: Jede Menge Operationen könnten Exceptions auslösen und es ist weit und breit kein `try/catch`-Block oder eine `throw()`-Spezifikation in Sicht.

☑ Richtlinie

Verstehen Sie die grundlegende, die hohe und die absolute Exception-Sicherheits-Garantie.

16. Wenn Sie jetzt der Meinung sind, es sei doch ziemlich kleinkariert, sich Sorgen darum zu machen, ob eine Nachricht vollständig ausgegeben wird oder nicht, dann haben Sie zum Teil Recht. In diesem Fall würde das tatsächlich niemanden kümmern. Die gleichen Grundsätze gelten jedoch für jede Funktion, die zwei Seiteneffekte auszuführen versucht, so dass die folgende Diskussion durchaus sinnvoll ist.

Um die hohe Garantie zu erfüllen, müssen entweder beide Effekte vollständig ausgeführt werden, oder es darf im Falle einer Exception kein einziger ausgeführt werden.

Können wir das erreichen? Ein möglicher Weg, es zu versuchen, sieht so aus:

```
// Versuch #1: Eine Verbesserung?
//
String EvaluateSalaryAndReturnName( Employee e )
{
  String result = e.First() + " " + e.Last();
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    String message = result + " ist überbezahlt\n";
    cout << message;
  }
  return result;
}
```

Das sieht schon nicht schlecht aus. Man beachte, dass wir das `endl` durch ein Newline-Zeichen ersetzt haben (was nicht exakt äquivalent ist), um den gesamten String mit einem einzigen Aufruf von `operator<<` ausgeben zu können. (Dadurch können wir natürlich immer noch nicht verhindern, dass das zugrunde liegende Stream-System während der Ausgabe fehlschlägt, was zu einer unvollständigen Ausgabe der Meldung führen würde. Aber zumindest haben wir alles getan, was wir auf dieser Abstraktionsebene tun können.)

Wie der folgende Anwendungscode verdeutlicht, wäre da noch eine Kleinigkeit:

```
// Ein Problem...
//
String theName;
theName = EvaluateSalaryAndReturnName( bob );
```

Da das Ergebnis per Wert zurückgegeben wird, kommt der Copy-Konstruktor von `String` zum Einsatz. Weiterhin erfordert das Kopieren des Ergebnisses nach `theName` den Copy-Zuweisungsoperator. Scheitert eine der Kopien, dann hat die Funktion zwar beide Seiteneffekte ausgeführt (schließlich wurde die Meldung komplett ausgegeben und der Rückgabewert vollständig konstruiert), aber das Ergebnis geht unwiederbringlich verloren.

Können wir das vielleicht durch Vermeiden der Kopie irgendwie besser machen? Wir könnten die Funktion zum Beispiel so abändern, dass sie zusätzlich eine nicht konstante `String`-Referenz übernimmt, in die der Rückgabewert platziert wird.

```
// Versuch #2: Besser?
//
void EvaluateSalaryAndReturnName( Employee e,
  String&  r )
{
```

```
  String result = e.First() + " " + e.Last();
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    String message = result + " ist überbezahlt\n";
    cout << message;
  }
  r = result;
}
```

Das mag besser aussehen, ist es aber nicht, da die Zuweisung zu `r` immer noch fehlschlagen kann, wodurch dann einer der Seiteneffekte ausgeführt worden wäre, der andere jedoch nicht.

Eine Möglichkeit, dieses Problem zu lösen, besteht darin, einen Zeiger auf einen dynamisch allozierten `String` zurückzugeben. Aber am besten ist, wir gehen noch einen Schritt weiter und geben den Zeiger in einem `auto_ptr` zurück.

```
// Versuch #3: Korrekt (endlich!).
//
auto_ptr<String>
EvaluateSalaryAndReturnName( Employee e )
{
  auto_ptr<String> result
      = new String( e.First() + " " + e.Last() );
  if( e.Title() == "CEO" || e.Salary() > 100000 )
  {
    String message = (*result) + " ist überbezahlt\n";
    cout << message;
  }
  return result;  // Auf Transfer der Eigentumsrechte
                  // verlassen, kein throw
}
```

Damit haben wir erfolgreich die gesamte Konstruktion des zweiten Seiteneffekts (des Rückgabewerts) verdeckt, während gleichzeitig sichergestellt ist, dass die Rückgabe nicht vor der Vollendung des ersten Seiteneffekts (der Ausgabe der Meldung) und nur durch Operationen erfolgt, die keine Exceptions werfen. Nach dem Ende der Funktion wird der Rückgabewert korrekt an den Aufrufer überstellt und auch in jeder denkbaren Situation wieder sauber freigegeben. Verwendet der Aufrufer den Rückgabewert, wird ihm durch Entgegennahme der `auto_ptr`-Kopie automatisch das Eigentum am Rückgabewert übertragen, andernfalls führt die Zerstörung des temporären `auto_ptr` automatisch auch zur Freigabe des allozierten `String`s, auf den er zeigt. Was ist der Preis für diese zusätzliche Sicherheit? Wie so oft bei der Implementierung der hohen Exception-Sicherheit kommt es auch hier zu einem (gewöhnlich unbedeutendem) Effizienzverlust in Form der zusätzlichen dynamischen Speicherallozierung. Wenn wir allerdings zwischen Effizienz und vorhersagbarem Verhalten und Richtigkeit abzuwägen haben, sollten wir die letzten beiden bevorzugen.

Betrachten wir nun Exception-Sicherheit im Zusammenhang mit mehrfachen Seiteneffekten. Für unserer Fallbeispiel stellte sich in Versuch #3 heraus, dass es sehr wohl möglich ist, beide Seiteneffekte mit Anvertrauen-oder-Zurücknehmen-Semantik zu realisieren (wenn man über die Stream-Problematik hinwegsieht). Der Grund dafür ist, dass sich die Möglichkeit ergab, beide Seiteneffekte atomar auszuführen, so dass die vorbereitende »wirkliche« Arbeit für beide so erledigt werden kann, dass für das eigentliche Sichtbarmachen der Seiteneffekte nur noch Operationen nötig sind, die keine Exceptions werfen.

Diesmal hatten wir zwar Glück, aber es ist nicht immer so einfach. Es ist nicht möglich, Funktionen zu schreiben, die die hohe Exception-Garantie erfüllen und trotzdem zwei oder mehr unabhängige Seiteneffekte aufweisen, die nicht atomar ausgeführt werden können (Was wäre zum Beispiel gewesen, wenn in unserem Beispiel die beiden Seiteneffekte darin bestanden hätten, eine Nachricht nach `cout` und eine andere nach `cerr` auszugeben?), denn die hohe Garantie verlangt, dass der Programmzustand im Falle einer Exception unverändert bleibt, dass also *keine* Seiteneffekte auftreten. Wenn in einer Funktion zwei Seiteneffekte nicht atomar realisiert werden können, dann besteht die einzige Möglichkeit, hohe Exception-Sicherheit herzustellen, gewöhnlich darin, diese eine Funktion in zwei Funktionen aufzuspalten, die für sich atomar ausführbar sind. Die Tatsache, dass sie nicht automatisch zusammen ausgeführt werden können, ist auf diese Weise zumindest am aufrufenden Code sichtbar.

☑ Richtlinie

Setzen Sie auf Kohäsion. Bemühen Sie sich immer, jedem Codeteil – jedem Modul, jeder Klasse und jeder Funktion – eine einzige, wohl definierte Aufgabe zu geben.

Zusammenfassung: Diese Lektion illustriert drei wichtige Dinge.

1. Für hohe Exception-Sicherheit bezahlt man häufig (jedoch nicht immer) mit Performance.
2. Eine Funktion mit mehreren, in keinem Zusammenhang stehenden Seiteneffekten kann nicht immer im Sinne der hohen Garantie exception-sicher gemacht werden. In einem solchen Fall hilft nur noch das Aufteilen der Funktion in mehrere Funktionen, deren Seiteneffekte dann atomar ausführbar sind.
3. Nicht alle Funktionen müssen hoch exception-sicher sein. Sowohl der ursprüngliche Code als auch Ansatz #1 genügen der grundlegenden Garantie. Ansatz #1 ist schon für viele Anwendungen ausreichend und minimiert die Wahrscheinlichkeit für Seiteneffekte während des Auftretens von Exceptions, ohne dass man dafür wie in Ansatz #3 mit Performance zu bezahlen hätte.

Zu dieser Lösung wäre bezüglich Streams und Seiteneffekten noch etwas anzumerken.

In der Aufgabenstellung dieser Lektion hieß es unter anderem: Nehmen Sie an, dass alle aufgerufenen Funktionen exception-sicher sind (Exceptions können ausgeworfen werden, es gibt dann aber keine Seiteneffekte), und dass alle verwendeten Objekte, auch die temporären, exception-sicher sind (Freigabe aller Ressourcen bei Zerstörung).

Wie sich herausstellte, kann unsere Annahme, keine der aufgerufenen Funktionen habe Seiteneffekte, nicht ganz richtig sein. Es gibt insbesondere keine Möglichkeit zu verhindern, dass die Stream-Operationen nach einer teilweisen Ausgabe scheitern. Daraus folgt, dass wir die geforderte Anvertrauen-oder-Zurücknehmen-Semantik bei jeder Funktion nicht erreichen können, die eine Ausgabe über Streams (zumindest über diese Standard-Streams) vornimmt.

Ein weiteres Problem ist die Tatsache, dass sich der Zustand eines Streams auch bei einer fehlschlagenden Ausgabe ändert. Wir haben diese Situation bisher nicht berücksichtigt, es ist jedoch möglich, die Funktion weiter zu verfeinern, so dass die Stream-Exceptions aufgefangen und die Error-Flags von `cout` zurückgesetzt werden, bevor die Exceptions zum Aufrufer weitergeleitet werden.

4 Klassendesign und Vererbung

Lektion 20: Klassenlehre	Schwierigkeitsgrad: 7

Wie gut kennen Sie sich mit den Einzelheiten des Klassenentwurfs aus? In dieser Lektion geht es nicht nur um offenkundige Fehler, sondern vor allem auch um professionellen Stil. Das Verständnis dieser Prinzipien wird Ihnen beim Entwurf robusterer und leichter zu wartender Klassen behilflich sein.

Sie führen eine Codeüberprüfung durch. Ein Programmierer hat die folgende Klasse geschrieben, bei der schlechter Stil und einige ernste Fehler zu bemängeln sind. Wie viele können Sie finden und wie würden Sie sie beheben?

```
class Complex
{
public:
  Complex( double real, double imaginary = 0 )
    : _real(real), _imaginary(imaginary)
  {
  }
  void operator+ ( Complex other )
  {
    _real = _real + other._real;
    _imaginary = _imaginary + other._imaginary;
  }
  void operator<<( ostream os )
  {
    os << "(" << _real << "," << _imaginary << ")";
  }
  Complex operator++()
  {
    ++_real;
    return *this;
  }
  Complex operator++( int )
  {
    Complex temp = *this;
    ++_real;
    return temp;
  }
```

```
private:
  double _real, _imaginary;
};
```

Lösung

Diese Klasse hat eine Menge Probleme – sogar mehr, als ich hier explizit zeigen werde. Die Bedeutung dieser Lektion besteht hauptsächlich darin, Regeln für den Klassenentwurf aufzuzeigen (zum Beispiel »Wie sieht die kanonische Form von `operator<<` aus?« und »Sollte `operator+` eine Elementfunktion sein?«), anstatt einfach nur die Stellen herauszusuchen, an denen die Schnittstelle schlecht konzipiert ist. Wie dem auch sei, beginnen werde ich mit der vielleicht gewinnbringendsten Feststellung: Warum eine Klasse für komplexe Zahlen schreiben, wenn es in der Standardbibliothek bereits eine gibt? Denn immerhin ist die dort vorhandene Version mit keinem der folgenden Probleme behaftet und wurde, basierend auf jahrelanger Praxis, von den besten Leuten unserer Industrie entworfen. Also nehmen Sie das bescheiden hin und verwenden Sie den Code wieder.

☑ Richtlinie

Verwerten Sie Code – speziell den der Standardbibliothek – wieder, anstatt eigenen zu schreiben. Ihr Programm wird dadurch schneller, einfacher und sicherer.

Die vielleicht beste Methode, um die Probleme in der `Complex`-Klasse zu lösen, besteht darin, sie erst gar nicht zu benutzen und stattdessen das `std::complex`-Template der Standardbibliothek einzusetzen.

Lassen Sie uns nun, da dies einmal gesagt wurde, die Klasse durchgehen und dabei die Fehler beheben.

Zuerst der Konstruktor:

1. Der Konstruktor erlaubt implizite Typumwandlung.

   ```
   Complex( double real, double imaginary = 0 )
     : _real(real), _imaginary(imaginary)
   {
   }
   ```

   Da für den zweiten Parameter ein Default-Wert angegeben ist, kann diese Funktion als Konstruktor mit einem Parameter benutzt werden und stellt damit eine implizite Typumwandlung von `double` nach `Complex` dar. In diesem Fall ist das wahrscheinlich auch ganz nützlich, aber wie wir schon in Lektion 6 gesehen haben, ist eine solche Umwandlung eventuell nicht immer beabsichtigt. Im Allgemeinen ist

es besser, Konstruktoren generell explizit zu machen, außer in den Fällen, in denen man die implizite Umwandlung bewusst erlauben will. (Siehe auch Lektion 39 zu impliziten Umwandlungen.)

☑ Richtlinie

Achten Sie auf versteckte temporäre Objekte, die durch implizite Umwandlungen erzeugt werden. Eine gute Möglichkeit, diese zu vermeiden, besteht darin, die Konstruktoren wenn möglich `explicit` *zu machen und Umwandlungsoperatoren zu vermeiden.*

2. `operator+` ist möglicherweise leicht ineffizient.

```
void operator+ (  Complex other  )
{
   _real = _real + other._real;
   _imaginary = _imaginary + other._imaginary;
}
```

Aus Effizienzgründen sollte der Parameter als konstante Referenz übergeben werden.

☑ Richtlinie

Übergeben Sie Objekte besser als `const&` *statt per Wert.*

Außerdem sollte in beiden Fällen »`a=a+b`« als »`a+=b`« geschrieben werden. In diesem speziellen Fall wird das natürlich nicht viel bringen, da nur `double`-Werte addiert werden, bei Klassentypen kann es aber einen signifikanten Unterschied ausmachen.

☑ Richtlinie

Schreiben Sie besser »`a op= b;`*« anstatt »*`a = a op b;`*« (wobei* `op` *für einen beliebigen Operator steht). Das sieht klarer aus und ist häufig auch effizienter.*

Die Funktion `operator+=` ist deshalb effizienter, weil sie mit dem im Ausdruck links stehenden Objekt direkt operiert und nur eine Referenz und kein temporäres Objekt zurückgibt. Betrachten Sie, um das zu sehen, die folgenden kanonischen Formen für `operator+=` und `operator+`, in denen diese Operatoren für gewöhnlich implementiert werden sollten.

```
T& T::operator+=( const T& other )
{
  //...
  return *this;
}
const T operator+( const T& a, const T& b )
{
  T temp( a );
  temp += b;
  return temp;
}
```

Beachten Sie die Verwandtschaft zwischen den Operatoren + und +=. Ersterer sollte durch Benutzung des Letzteren implementiert werden, und zwar sowohl der Einfachheit (der Code ist leichter zu schreiben) als auch der Konsistenz wegen (beide führen die gleiche Operation aus, bei späterer Wartung besteht weniger die Gefahr, dass sie voneinander abweichen).

☑ Richtlinie

Außer der einfachen Version eines Operators (zum Beispiel `operator+`) sollten Sie immer auch den entsprechenden Zuweisungsoperator (zum Beispiel `operator+=`) zur Verfügung stellen und dabei Ersteren durch Letzteren implementieren. Bewahren Sie auch immer die natürliche Verwandtschaft zwischen `op` und `op=` (wobei `op` für einen beliebigen Operator steht).

3. `operator+` sollte keine Elementfunktion sein.

```
void  operator+ ( Complex other )
{
  _real = _real + other._real;
  _imaginary = _imaginary + other._imaginary;
}
```

Wenn Sie implizite Typumwandlungen aus anderen Typen zulassen und `operator+` so wie hier eine Elementfunktion ist, wird er nicht in der natürlichen Weise funktionieren, die der Benutzer erwartet. Besonders wenn `Complex`-Objekte zu nummerischen Werten addiert werden sollen, kann man zwar »`a = b + 1.0`« aber nicht »`a = 1.0 + b`« schreiben, da ein als Elementfunktion realisierter `operator+` ein `Complex` auf der linken Seite des Ausdrucks erwartet (und kein `const double`).

Für die Addition von `Complex`-Objekten auf `double`s mag es außerdem sinnvoll sein, zusätzlich die überladenen Funktionen `operator+(const Complex&, double)` und `operator+(double, const Complex&)` zur Verfügung zu stellen.

☑ Richtlinie

Entscheiden Sie bevorzugt nach den folgenden Regeln, ob ein Operator als Elementfunktion oder nicht realisiert werden sollte:

- *`()[]` und `->` müssen Elementfunktionen sein.*
- *`>>` und `<<` sind nie Elementfunktionen (wenn nötig, `friend`s).*
- *Für alle anderen Operatoren gilt:*
  Muss die Funktion `virtual` sein, wird sie zur Elementfunktion,
  wird für das am weitesten links stehende Argument Typumwandlung benötigt, wird die Funktion zur Elementfunktion (und wenn nötig `friend`),
  kann sie nur durch Verwendung der öffentlichen Klassenschnittstelle implementiert werden, wird sie nicht zur Elementfunktion, in allen anderen Fällen wird sie zur Elementfunktion.

4. `operator+` sollte das Objekt nicht verändern sowie ein temporäres Objekt zurückgeben, das die Summe enthält.

```
void  operator+ ( Complex other )
{
  _real   = _real + other._real;
  _imaginary   = _imaginary + other._imaginary;
}
```

   Man beachte, dass der Rückgabetyp »`const Complex`« und nicht nur »`Complex`« lauten sollte, damit Ausdrücke wie »`a+b=c`« ungültig bleiben.

5. Wenn Operator `op` definiert ist, sollte das normalerweise auch für Operator `op=` der Fall sein. Hier sollte aus diesem Grund `operator+=` definiert werden, da `operator+` ja schließlich auch definiert ist. Weiterhin sollte Letzterer durch Ersteren implementiert werden.

6. `operator<<` sollte keine Elementfunktion sein.

```
void   operator  (   ostream os   )
{
  os << "(" << _real << "," << _imaginary << ")";
}
```

   Hier gilt im Prinzip das Gleiche wie bei `operator+`. Die Parameter sollten hier »`(ostream&,constComplex&)`« lauten. Der `operator<<` als Nichtelementfunktion sollte immer mit Hilfe einer (häufig virtuellen) Elementfunktion, die gewöhnlich `Print()` o. Ä. heißt, implementiert werden.

   Um den allgemeinen Konventionen zu entsprechen, sollte ein realer operator<< zusätzlich noch zum Beispiel die aktuellen Format-Flags des Streams prüfen. Details

dazu können Sie Ihrem Lieblingsbuch über die Standardbibliothek bzw. über iostreams entnehmen.

7. Weiterhin sollte `operator<<` eine Referenz auf den Stream, also ein »`ostream&`« zurückgeben, um Kettenbildung zu vermeiden. Dadurch kann dann der Benutzer in natürlicher Weise Code wie »`cout << a << b;`« schreiben.

☑ Richtlinie

Lassen Sie `operator<<` und `operator>>` immer Stream-Referenzen zurückgeben.

8. Der Preinkrementoperator hat einen falschen Rückgabetyp.

```
Complex operator++()
{
  ++_real;
  return *this;
}
```

Preinkrement sollte eine nicht konstante Referenz zurückgeben, also in diesem Fall `Complex&`. Das vermeidet unnötige Ineffizienz und ist für den Anwender intuitiver.

9. Der Postinkrementoperator hat einen falschen Rückgabetyp.

```
Complex operator++( int )
{
  Complex temp = *this;
  ++_real;
  return *temp;
}
```

Postinkrement sollte einen konstanten Wert zurückgeben, also in diesem Fall ein `const Complex`. Indem wir keine Änderungen am zurückgegebenen Objekt erlauben, verhindern wir fragwürdigen Code wie »`a++++`«, der nicht das tut, was ein naiver Anwender erwarten würde.

10. Postinkrement sollte durch Preinkrement ausgedrückt werden. Siehe Lektion 6 zur kanonischen Postinkrement-Form.

☑ Richtlinie

Implementieren Sie zur Wahrung der Konsistenz Postinkrement immer durch Preinkrement, andernfalls erhalten Ihre Anwender überraschende (und oft unerfreuliche) Ergebnisse.

11. Vermeiden Sie reservierte Namen.

```
private:
  double _real, _imaginary;
```

Populäre Werke wie *Design Patterns* (Gamma95) benutzen zwar führende Unterstriche in Variablennamen, aber Sie sollten das nicht tun. Im Standard sind einige Bezeichner mit führenden Unterstrichen zu Implementierungszwecken reserviert. Die entsprechenden Regeln sind sowohl für Sie als auch für die Compilerhersteller schwierig zu merken, daher sollten Sie möglicherweise vollständig auf führende Unterstriche verzichten.[1] Ich selbst bevorzuge stattdessen die Konvention, jede Elementvariable mit einem Unterstrich am Ende zu kennzeichnen.

Die korrigierte Version der Klasse, allerdings ohne Design- und Stilverbesserungen, die bis hier nicht explizit erwähnt wurden, sieht so aus:

```
class Complex
{
public:
  explicit Complex( double real, double imaginary = 0 )
    : real_(real), imaginary_(imaginary)
  {
  }

  Complex& operator+=( const Complex& other )
  {
    real_ += other.real_;
    imaginary_ += other.imaginary_;
    return *this;
  }

  Complex& operator++()
  {
    ++real_;
    return *this;
  }

  const Complex operator++( int )
  {
    Complex temp( *this );
    ++*this;
```

1. Technisch gesehen sind Namen mit führenden Unterstrichen nur für Nichtelementbezeichner reserviert, so dass es scheinbar kein Problem darstellt, derartige Namen innerhalb von Klassendeklarationen zu verwenden. Das trifft allerdings in der Praxis nicht ganz zu, da einige Implementierungen mittels `#define` Makros definieren, die keinerlei Namensräume beachten und damit sowohl mit Elementbezeichnern als auch mit Nichtelementbezeichnern kollidieren. Es ist daher wesentlich einfacher, auf führende Unterstriche zu verzichten und damit den ganzen Zank zu umgehen.

```
    return temp;
  }

  ostream& Print( ostream& os ) const
  {
    return os << "(" << real_ << "," << imaginary_ << ")";
  }

private:
  double real_, imaginary_;
};

const Complex operator+( const Complex& lhs, const Complex& rhs )
{
  Complex ret( lhs );
  ret += rhs;
  return ret;
}

ostream& operator<<( ostream& os, const Complex& c )
{
  return c.Print(os);
}
```

Lektion 21: Überladen von virtuellen Funktionen Schwierigkeitsgrad: 6

Virtuelle Funktionen sind ein ziemlich grundlegendes Merkmal, stellen allerdings dem Unvorsichtigen gelegentlich subtile Fallen. Wenn Sie Fragen wie die folgende beantworten können, beherrschen Sie virtuelle Funktionen aus dem Effeff und laufen nicht Gefahr, Unmengen von Zeit beim Debuggen von Problemen wie dem unten gezeigten zu vergeuden.

In Ihrer Firma untersuchen Sie die staubigen Ecken des Codearchivs. Dabei entdecken Sie das folgende Programmfragment, das von einem unbekannten Programmierer geschrieben wurde. Es scheint so, als hätte dieser Programmierer mit einigen C++-Merkmalen experimentiert. Welche Programmausgabe hat er wahrscheinlich erwartet und wie sieht das tatsächliche Ergebnis aus?

```
#include <iostream>
#include <complex>
using namespace std;
class Base
{
public:
  virtual void f( int );
  virtual void f( double );
```

```
  virtual void g( int i = 10 );
};
void Base::f( int )
{
  cout << "Base::f(int)" << endl;
}
void Base::f( double )
{
  cout << "Base::f(double)" << endl;
}
void Base::g( int i )
{
  cout << i << endl;
}
class Derived: public Base
{
public:
  void f( complex<double> );
  void g( int i = 20 );
};
void Derived::f( complex<double> )
{
  cout << "Derived::f(complex)" << endl;
}
void Derived::g( int i )
{
  cout << "Derived::g() " << i << endl;
}
void main()
{
  Base    b;
  Derived d;
  Base*   pb = new Derived;
  b.f(1.0);
  d.f(1.0);
  pb->f(1.0);
  b.g();
  d.g();
  pb->g();
  delete pb;
}
```

Lösung

Lassen Sie uns zuerst einige stilistische Aspekte und einen wirklichen Fehler betrachten.

1. »`void main()`« entspricht nicht dem Standard und ist damit unportabel.

   ```
   void main()
   ```

   Ja, ich weiß, dass das leider in vielen Büchern auftaucht. Einige Autoren meinen sogar, »`void main()`« wäre standardkonform. Ist es aber nicht. War es auch nie, nicht einmal in den 70ern im ursprünglichen C (vor dem Standard).

   Obwohl »`void main()`« keine der legalen Deklarationen von `main()` darstellt, lassen es viele Compiler dennoch zu. Das bedeutet aber nicht, dass, auch wenn es heute mit Ihrem aktuellen Compiler funktioniert, es bei einem neuen Compiler zwangsläufig ebenso klappt. Man gewöhnt sich besser an, eine der portablen Standarddeklarationen von `main()` zu benutzen:

   ```
   int main()
   int main( int argc, char* argv[] )
   ```

   Sie haben vielleicht auch schon bemerkt, dass das vorgestellte Programm in `main()` keine `return`-Anweisung ausführt. Das ist aber auch bei einer dem Standard entsprechenden `main`-Funktion kein Problem (obwohl es bestimmt von gutem Stil zeugt, dem Aufrufer eventuelle Fehler anzuzeigen), da eine fehlende `return`-Anweisung den gleichen Effekt hat wie »`return 0;`«. Aber Vorsicht: Zum Zeitpunkt, da dies geschrieben wird, implementieren viele Compiler diese Regel noch nicht und geben bei fehlender `return`-Anweisung eine entsprechende Warnung aus.

2. »`delete pb;`« ist unsicher.

   ```
   delete pb;
   ```

   Das sieht harmlos aus. Wäre es auch, wenn der Autor von `Base` es nicht versäumt hätte, einen virtuellen Destruktor zur Verfügung zu stellen. In solch einem Fall ist das Löschen über einen Zeiger auf die Basis nicht zu empfehlen, da dann der falsche Destruktor aufgerufen und `delete` eine falsche Objektgröße übergeben wird, was wiederum die Speicherverwaltung korrumpiert.

☑ Richtlinie

Machen Sie den Destruktor jeder Basisklasse virtuell, es sei denn, Sie wissen genau, dass nie jemand eine Instanz einer davon abgeleiteten Klasse über einen Zeiger auf die Basis zu löschen versucht.

Für die folgenden Punkte ist es wichtig, zwischen den drei folgenden Bezeichnungen zu unterscheiden:

- Eine Funktion `f()` zu *überladen* bedeutet, eine andere Funktion mit dem gleichen Namen (`f`), aber mit anderen Argumenten im gleichen Namensraum zu platzieren. Wenn `f` tatsächlich aufgerufen wird, versucht der Compiler anhand der übergebenen Parameter, die am besten passende Funktion zu finden.

- Eine virtuelle Funktion `f()` zu *überschreiben* bedeutet, in einer abgeleiteten Klasse eine andere Funktion mit dem gleichen Namen (`f`) und den gleichen Argumenten zur Verfügung zu stellen.
- Eine Funktion `f()` in einem umgebenden Namensraum (Basisklasse, umgebende Klasse oder `namespace`) zu *verstecken* bedeutet, eine andere Funktion mit dem gleichen Namen (`f`) in einen inneren Namensraum (abgeleitete Klasse, verschachtelte Klasse oder `namespace`) zu platzieren, wodurch der Name der Funktion im äußeren Namensraum überdeckt wird.

3. `Derived::f` stellt keine Überladung dar.

   ```
   void Derived::f( complex<double> )
   ```

   In `Derived` werden die `Base::f()`-Funktionen nicht überladen, sondern versteckt. Diese Unterscheidung ist wichtig, denn so sind `Base::f(int)` und `Base::f(double)` im Namensraum von `Derived` nicht sichtbar. (Erstaunlicherweise geben bestimmte Compiler in einem solchen Fall nicht einmal eine Warnung aus, so dass es umso wichtiger ist, darüber Bescheid zu wissen.)

   Das stellt natürlich keinen Fehler dar, wenn der Autor von `Derived` ganz bewusst `Base`-Funktionen namens `f` verstecken wollte.[2] Für gewöhnlich geschieht eine solche Namensüberdeckung allerdings unabsichtlich und ist dann recht überraschend. Die korrekte Methode, um die Namen in den Namensraum von `Derived` zu bringen, besteht in der Deklaration »`using Base::f;`«.

☑ Richtlinie

Wenn Sie eine Funktion schreiben, die den gleichen Namen hat wie eine geerbte Funktion, dann sorgen Sie mit einer `using`-Deklaration dafür, dass die geerbten Funktionen sichtbar werden, falls Sie sie nicht verstecken wollen.

4. `Derived::g` überschreibt `Base::g`, ändert aber den Default-Parameter.

   ```
   void g( int i  = 20  )
   ```

   Das ist ausgesprochen benutzerunfreundlich. Verändern Sie beim Überschreiben einer geerbten Funktion nicht den Default-Parameter, es sein denn, Sie wollen wirklich Verwirrung stiften. (Im Allgemeinen ist Überschreiben den Default-Parametern vorzuziehen, aber das ist ein Thema für sich.) Ja, es handelt sich dabei um gültiges C++; und ja, das Ergebnis ist genau definiert; und nein, tun Sie es nicht.

   Welche Konfusion man damit verursachen kann, werden wir im Weiteren noch sehen.

2. Wenn ein Basisname absichtlich versteckt werden soll, verpasst man dem Namen besser eine private `using`-Deklaration.

☑ Richtlinie

Verändern Sie niemals den Default-Parameter einer überschriebenen geerbten Funktion.

Lassen Sie uns nun, nachdem diese Dinge abgehakt sind, einen Blick auf das Hauptprogramm werfen und prüfen, ob es die Absichten des Programmierers umsetzt.

```
void main()
{
  Base    b;
  Derived d;
  Base*   pb = new Derived;
  b.f(1.0);
```

Kein Problem. Dieser erste Aufruf führt wie erwartet auf `Base::f(double)`.

```
  d.f(1.0);
```

Hierdurch wird `Derived::f(complex<double>)` aufgerufen. Warum? Nun, in `Derived` fehlt die Deklaration »`using Base::f;`«, so dass `Base::f(int)` und `Base::f(double)` natürlich nicht aufgerufen werden können. Sie sind nicht im gleichen Namensraum wie `Derived::f(complex<double>)` und werden daher zum Überladen nicht herangezogen.

Der Programmierer könnte den Aufruf von `Base::f(double)` erwartet haben. Es kommt in diesem Fall jedoch trotzdem zu keiner Fehlermeldung, denn es existiert zum Glück (?) eine implizite Umwandlung von `double` nach `complex<double>`. Der Aufruf wird daher vom Compiler als `Derived::f(complex<double>(1.0))` verstanden.

```
  pb->f(1.0);
```

Obwohl der `Base*`-Zeiger `pb` auf eine `Derived`-Instanz zeigt, ruft diese Anweisung interessanterweise `Base::f(double)` auf, denn der Aufruf einer überladenen Funktion wird vom Compiler anhand des statischen (hier `Base`) und nicht des dynamischen (hier `Derived`) Typs aufgelöst.

Aus dem gleichen Grund kann der Aufruf »`pb->f(complex<double>(1.0));`« nicht übersetzt werden, da in der `Base`-Schnittstelle keine passende Funktion existiert.

```
  b.g();
```

Diese Anweisung gibt »`10`« aus, da einfach die Funktion `Base::g(int)` aufgerufen wird, deren Default-Parameter 10 beträgt. Kein Problem.

```
  d.g();
```

Diese Anweisung gibt »`Derived::g() 20`« aus, da einfach die Funktion `Derived::g(int)` aufgerufen wird, deren Default-Parameter 20 beträgt. Auch kein Problem.

```
  pb->g();
```

Hierdurch wird »`Derived::g() 10`« ausgegeben.

»Moment mal«, protestieren Sie jetzt, »was geht hier vor?« Dieses Ergebnis verblüfft Sie jetzt vielleicht, aber Sie werden erkennen, dass der Compiler vollkommen richtig gehandelt hat.[3] Default-Parameter werden genau wie beim Überladen dem statischen Typ (hier `Base`) entnommen, so dass der Wert `10` zum Einsatz kommt. Die Funktion ist nun aber virtuell, weswegen die aufzurufende Funktion zur Laufzeit anhand des dynamischen Typs (hier `Derived`) festgestellt wird.

Wenn Sie die letzten Absätze über das Verstecken von Bezeichnern und über die Bedingungen, unter denen statische bzw. dynamische Typen benutzt werden, verstanden haben, dann sind Sie jetzt Experte bezüglich dieser Thematik. Gratulation!

```
  delete pb;
}
```

Wie bereits angemerkt, hinterlässt dieses `delete` Schäden in der Speicherverwaltung und führt zu einer nur teilweisen Objektzerstörung; siehe dazu den obigen Abschnitt über virtuelle Destruktoren.

Lektion 22: Klassenbeziehungen – Teil 1 Schwierigkeitsgrad: 5

Wie gut sind Sie beim objektorientierten Entwurf? Diese Lektion veranschaulicht einen oft begangenen Klassen-Design-Fehler, der nach wie vor vielen Programmierern unterläuft.

Eine Netzwerkanwendung arbeite mit zwei Arten von Netzwerkverbindungen, jede mit ihrem eigenen Protokoll. Zwischen diesen beiden Protokollen gebe es Ähnlichkeiten (einige Berechnungen und sogar einige Nachrichten sind gleich), so dass der Programmierer auf die Idee kam, diese Gemeinsamkeiten mit dem folgenden Entwurf in einer Klasse `BasicProtocol` zu kapseln.

```
class BasicProtocol /* : mögliche Basisklassen */
{
public:
  BasicProtocol();
  virtual ~BasicProtocol();
  bool BasicMsgA( /*...*/ );
  bool BasicMsgB( /*...*/ );
  bool BasicMsgC( /*...*/ );
};
class Protocol1 : public BasicProtocol
{
public:
  Protocol1();
  ~Protocol1();
  bool DoMsg1( /*...*/ );
```

3. Obwohl man den Programmierer von `Derived` dafür natürlich raus auf den Parkplatz zerren und kräftig anbrüllen sollte.

```
    bool DoMsg2( /*...*/ );
    bool DoMsg3( /*...*/ );
    bool DoMsg4( /*...*/ );
};

class Protocol2 : public BasicProtocol
{
public:
    Protocol2();
    ~Protocol2();
    bool DoMsg1( /*...*/ );
    bool DoMsg2( /*...*/ );
    bool DoMsg3( /*...*/ );
    bool DoMsg4( /*...*/ );
    bool DoMsg5( /*...*/ );
};
```

Jede `DoMsg...()`-Elementfunktion ruft je nach Bedarf die `BasicProtocol::Basic...()`-Funktionen auf, um die häufig zu verrichtenden Arbeiten auszuführen. Die eigentliche Übertragung wird jedoch von den `DoMsg...()`-Funktionen selbst durchgeführt. Zu jeder Klasse könnten noch weitere Elemente gehören, aber wir nehmen an, dass alle relevanten Elemente aufgelistet sind.

Kommentieren Sie diesen Entwurf. Gibt es etwas, das Sie ändern würden? Wenn ja, was?

Lösung

Diese Lektion illustriert einen zu häufig auftretenden Fallstrick beim objektorientierten Design von Klassenbeziehungen. Fassen wir kurz zusammen: Die Klassen `Protocol1` und `Protocol2` sind von einer gemeinsamen Basisklasse namens `BasicProtocol` abgeleitet, und zwar `public`, die die allen gemeinsame Arbeit ausführt.

Der Schlüssel zum Erkennen des Problems sind die beiden folgenden Sätze:

> *Jede* `DoMsg...()`*-Elementfunktion ruft je nach Bedarf die* `BasicProtocol::Basic...()`*-Funktionen auf, um die häufig zu verrichtenden Arbeiten auszuführen. Die eigentliche Übertragung wird jedoch von den* `DoMsg...()`*-Funktionen selbst durchgeführt.*

Hierdurch wird eindeutig eine »ist-implementiert-mit«-Beziehung beschrieben, was in C++ entweder »private Vererbung« oder »Realisierung als Elementvariable« (*membership*) bedeutet. Leider missverstehen das viele Leute immer noch häufig als »öffentliche Vererbung« und verwechseln damit die Vererbung der Implementierung mit der Vererbung der Schnittstelle, was nicht das Gleiche bedeutet. Hier ist diese Verwechslung genau die Ursache des Problems.

☒ Typischer Fehler

Setzen Sie niemals öffentliche Vererbung ein, es sei denn, Sie wollen wirklich eine IST-EINE- und FUNKTIONIERT-WIE-Beziehung nach Liskov modellieren. Alle überschriebenen Elementfunktionen dürfen nicht mehr erfordern und nicht weniger versprechen.

Übrigens landen Programmierer, die sich diesen Fehler (also öffentliche Vererbung zur Implementierung zu verwenden) angewöhnt haben, gewöhnlich in tiefen Klassenhierarchien. Durch die dadurch entstehende unnötige, zusätzliche Komplexität steigt der Wartungsaufwand erheblich. Außerdem werden die Anwender dazu gezwungen, sich mit den Schnittstellen vieler Klassen zu befassen, auch wenn sie lediglich eine bestimmte abgeleitete Klasse benutzen wollen. Weiterhin können Speicherverbrauch und Performance des Programms beeinflusst werden, wenn unnötige *vtables* (Tabellen zur Auflösung virtueller Funktionsaufrufe) sowie indirekte Verweise zu Klassen hinzukommen, die diese gar nicht benötigen. Wenn Sie sich selbst wiederholt dabei ertappen, wie Sie tiefe Vererbungshierarchien erzeugen, dann sollten Sie Ihren Entwurfstil daraufhin überprüfen, ob Sie sich diese schlechte Angewohnheit zu Eigen gemacht haben. Tief gehende Hierarchien werden selten benötigt und sind fast nie sinnvoll. Und wenn Sie das nicht glauben, sondern stattdessen denken: »Objekorientierte Programmierung ist nicht objektorientiert ohne einen Haufen Vererbungen«, dann betrachten Sie als gutes Gegenbeispiel die Standardbibliothek.

☑ Richtlinie

Erben Sie niemals öffentlich, um Code wiederzuverwenden (aus der Basisklasse). Erben Sie nur öffentlich, um selbst wiederverwendet zu werden (von Code, der die Basisobjekte polymorph benutzt)[1].

Um die Sache etwas detaillierter zu betrachten seien hier mehrere Hinweise gegeben, mit Hilfe derer man das Problem erkennen kann.

1. Es gibt keine virtuellen Funktionen in `BasicProtocol` (außer dem Destruktor, zu dem wir gleich kommen).[5] Diese Klasse ist also gar nicht dazu gedacht, polymorph benutzt zu werden. Das ist ein guter Hinweis gegen öffentliche Vererbung.

2. `BasicProtocol` hat keine `protected`-Elemente. Das bedeutet, dass es keine »Schnittstelle für Ableitungen« gibt, was ein guter Hinweis gegen jegliche Vererbung insgesamt ist, ob nun öffentlich oder privat.

4. Dank an Marshall Cline, den Koautor des Klassikers *C++ FAQs* (Cline 95), für diese Richtlinie.
5. Auch wenn `BasicProtocol` selbst wieder von einer anderen Klasse abgeleitet wäre, kämen wir zu der gleichen Schlussfolgerung, denn die Klasse fügt keine neuen virtuellen Funktionen hinzu. Gäbe es in irgendeiner der Basisklassen virtuelle Funktionen, so wäre es diese entfernte Basisklasse, die polymorph benutzt werden könnte, nicht `BasicProtocol`. Wenn überhaupt, dann sollten wir also von dieser entfernten Basisklasse ableiten.

3. `BasicProtocol` kapselt gemeinsame Funktionalität, führt aber wie beschrieben im Gegensatz zu den abgeleiteten Klassen selbst keine Übertragungen durch. Das heißt, dass eine `BasicProtocol`-Instanz weder WIE EINE Instanz der abgeleiteten Klassen FUNKTIONIERT, noch WIE EIN abgeleitetes Protokollobjekt BENUTZBAR ist. Mit öffentlicher Vererbung sollte nur eins modelliert werden – die IST-EIN-Beziehung, die dem Liskov-Substitutionsprinzip folgt. Der größeren Klarheit wegen nenne ich das gewöhnlich FUNKTIONIERT-WIE-EIN oder BENUTZBAR-WIE-EIN.[6]

4. Die abgeleiteten Klassen nutzen lediglich die öffentliche Schnittstelle von `BasicProtocol` und ziehen damit keinen Nutzen daraus, abgeleitete Klassen zu sein. Sie könnten daher ihre Arbeit auch über ein separates Hilfsobjekt vom Typ `BasicProtocol` erledigen.

   Damit verbleiben noch einige Schönheitskorrekturen. Da `BasicProtocol` nicht dazu bestimmt ist, als Basisklasse für Ableitungen zu fungieren, ist hier der virtuelle Destruktor unnötig (stattdessen sogar irreführend) und sollte entfernt werden. Außerdem wäre es wahrscheinlich günstig, `BasicProtocol` etwa in `MessageCreator` oder `MessageHelper` umzubenennen.

   Wie sollten wir nach diesen Änderungen die »ist-implementiert-mit«-Beziehung modellieren – mit privater Vererbung oder mit Realisierung als Elementvariable (*membership*)? Die Antwort darauf kennen wir schon:

☑ Richtlinie

Bevorzugen Sie bei der Modellierung von »ist implementiert mit« immer Realisierung als Elementvariable (membership)/Aggregation, nicht Vererbung. Benutzen Sie private Vererbung nur dann, wenn sie absolut nötig ist – also wenn Sie Zugriff auf `protected`*-Elemente brauchen oder virtuelle Funktionen überschreiben wollen. Benutzen Sie niemals öffentliche Vererbung, um Code wiederzuverwenden.*

Mit der Realisierung als Elementvariable (*membership*) erreicht man eine bessere Trennung der Zuständigkeiten, da die benutzende Klasse als normaler Anwender auftritt, der nur Zugriff auf die öffentliche Schnittstelle der benutzten Klasse hat. Wenn Sie diese Methode vorziehen, werden Sie sehen, dass Ihr Code klarer, einfacher zu lesen und einfacher zu warten sein wird. Oder kurz gesagt: Ihr Code wird dann weniger kosten.

6. Natürlich nimmt man manchmal beim öffentlichen Erben einer Schnittstelle auch einen Teil der Implementierung mit, wenn in der Basisklasse sowohl die gewünschte Schnittstelle als auch eigene Implementierung vorhanden ist. Das kann man aber praktisch immer "wegdesignen« (siehe nächste Lektion). Es ist jedoch nicht immer notwendig, dem puristischen Anspruch nach "einer Verantwortlichkeit pro Klasse« zu genügen.

Lektion 23: Klassenbeziehungen – Teil 2 Schwierigkeitsgrad: 6

Entwurfsmuster sind ein wichtiges Werkzeug beim Schreiben wiederverwendbaren Codes. Erkennen Sie die Muster, die für diese Lektion verwendet wurden? Wenn ja, können Sie sie verbessern?

Datenbankprogramme müssen häufig eine bestimmte Funktion auf jeden Datensatz (oder auf ausgewählte Datensätze) einer Tabelle anwenden. Dabei durchlaufen sie zunächst die gesamte Tabelle nur lesend, um festzustellen, welche Datensätze zu bearbeiten sind. Bei einem zweiten Durchlauf werden dann die eigentlichen Änderungen vorgenommen.

Um das ständige Neuprogrammieren dieses Algorithmus' zu vermeiden, habe ein Programmierer versucht, mit der folgenden abstrakten Klasse ein generisches wiederverwendbares Framework zu schaffen. Die Idee dabei ist, die sich wiederholenden Arbeiten durch die Klasse kapseln zu lassen, indem zuerst eine Liste der zu bearbeitenden Tabellenzeilen kompiliert und anschließend die gewünschte Operation auf diese Zeilen angewendet wird. Die Operationen selbst werden durch die abgeleiteten Klassen festgelegt.

```
//-------------------------------------------------
// Datei gta.h
//-------------------------------------------------
class GenericTableAlgorithm
{
public:
  GenericTableAlgorithm( const string& table );
  virtual ~GenericTableAlgorithm();

  // Process() gibt bei Erfolg true zurück.
  // Erledigt die gesamte Arbeit: a) liest die Daten-
  // sätze der Tabelle, ruft für jeden Filter() auf, um
  // festzustellen, ob er zu den zu bearbeitenden
  // Zeilen zählt; b) ruft für jede dieser Zeilen
  // ProcessRow() auf, wenn die Liste der betroffenen
  // Zeilen fertig ist.
  //
  bool Process();

private:
  // Filter() gibt true zurück, wenn die Zeile zu
  // den zu bearbeitenden Zeilen zählen soll. Die
  // Default-Rückgabe ist true (zählt jede Zeile
  // dazu).
  //
  virtual bool Filter( const Record& );

  // ProcessRow() wird für jeden Datensatz, der
```

```
  // bearbeitet werden soll, einmal aufgerufen. Hier
  // wird die spezialisierte Funktionalität der
  // konkreten Klassen platziert. (Daraus folgt, dass
  // jede zu bearbeitende Zeile zweimal gelesen wird.
  // Annahme: Dies ist notwendig und für die
  // Effizienz nicht relevant.)
  //
  virtual bool ProcessRow( const PrimaryKey& ) =0;

  class GenericTableAlgorithmImpl* pimpl_; // MYOB
};
```

Die Ableitung einer konkreten Klasse und deren Benutzung sieht beispielsweise so aus:

```
class MyAlgorithm : public GenericTableAlgorithm
{
  // ... überschreibe Filter() und ProcessRow() zur
  //     Implementierung einer spezifischen Operation ...
};
int main()
{
  MyAlgorithm a( "Kunde" );
  a.Process();
}
```

Die Fragen dazu lauten:

1. Es handelt sich hierbei um ein gutes Design, das ein wohlbekanntes Entwurfsmuster implementiert. Wie heißt dieses Entwurfsmuster und warum ist es hier sinnvoll?

2. Untersuchen Sie kritisch und ohne Veränderung des zugrunde liegenden Entwurfs die Art und Weise, in der dieses Design ausgeführt wurde. Was hätten Sie anders gemacht? Welchen Zweck hat das `pimpl_`-Element?

3. Dieses Design kann in der Tat beträchtlich verbessert werden. Wie lauten die Zuständigkeiten von `GenericTableAlgorithm`? Wenn es mehr als eine sind, wie könnten sie besser gekapselt werden? Erläutern Sie, wie Ihre Antwort die Wiederverwendbarkeit und insbesondere die Erweiterbarkeit der Klasse beeinflusst.

Lösung

Betrachten wir die Fragen der Reihe nach.

1. Es handelt sich hierbei um ein gutes Design, das ein wohlbekanntes Entwurfsmuster implementiert. Wie heißt dieses Entwurfsmuster und warum ist es hier sinnvoll?

Es ist das Template-Methoden-Muster (*template method pattern*, Gamme95), nicht zu verwechseln mit C++-Templates. Sein Einsatz ist hier sinnvoll, da wir damit die Art und Weise von etwas, das stets nach dem gleichen Schema erfolgt, verallgemeinern können. Unterschiede gibt es nur bei den Details, und dafür ist dann die abgeleitete Klasse zuständig. Zusätzlich kann die abgeleitete Klasse das Template-Methoden-Muster erneut anwenden – also die virtuelle Funktion als Wrapper um eine neue virtuelle Funktion überschreiben – so dass verschiedene Teile des Schemas in verschiedenen Ebenen der Klassenhierarchie realisiert werden können.

(Hinweis: Oberflächlich gesehen hat das Pimpl-Idiom Ähnlichkeit mit dem Bridge-Idiom, hat hier aber nur die Aufgabe, als eine Art Firewall gegen Übersetzungsabhängigkeiten die Implementierung dieser speziellen Klasse zu verbergen. Es fungiert nicht wirklich als eine erweiterbare Brücke (*bridge*). Mit dem Pimpl-Idiom beschäftigen wir uns eingehend in Lektionen 26 bis 30.)

☑ Richtlinie

Vermeiden Sie öffentliche virtuelle Funktionen. Benutzen Sie stattdessen besser das Template-Methoden-Muster.

☑ Richtlinie

Lernen und benutzen Sie Entwurfsmuster.

2. Untersuchen Sie kritisch und ohne Veränderung des zugrunde liegenden Entwurfs die Art und Weise, in der dieses Design ausgeführt wurde. Was hätten Sie anders gemacht? Welchen Zweck hat das `pimpl_`-Element?

   Dieser Entwurf benutzt `bool` für die Rückgabewerte. Offensichtlich gibt es keinen anderen Weg (Statuscodes oder Exceptions), um Fehler zu melden. Das mag, abhängig von den gestellten Anforderungen, ausreichend sein, muss aber trotzdem erwähnt werden.

   Das (absichtlich aussprechbare) `pimpl_`-Element verbirgt sehr schön die Implementierung hinter einem Zeiger. Die Struktur, auf die `pimpl_` zeigt, enthält die privaten Elementfunktionen und Elementvariablen, so dass bei diesbezüglichen Änderungen der anwendende Code nicht ebenfalls neu übersetzt werden muss. Diese wichtige, u.a. von Lakos (Lakos96) beschriebene Methode ist zwar beim Programmieren ein wenig lästig, kompensiert aber zum Teil die Tatsache, dass C++ über kein Modulsystem verfügt.

☑ Richtlinie

Bevorzugen Sie für oft benutzte Klassen das Compiler-Firewall-Idiom (Pimpl-Idiom), um die Implementierungsdetails zu verbergen. Benutzen Sie einen undurchsichtigen Zeiger (einen Zeiger auf eine deklarierte aber undefinierte Klasse), der als »`struct XxxxImpl* pimpl_;`« deklariert ist, um die privaten Elemente (sowohl Statusvariablen als auch Elementfunktionen) zu speichern. Beispiel: »`class Map { private: struct MapImpl* pimpl_; };`«.

3. Dieses Design kann in der Tat beträchtlich verbessert werden. Wie lauten die Zuständigkeiten von `GenericTableAlgorithm`? Wenn es mehr als eine sind, wie könnten sie besser gekapselt werden? Erläutern Sie, wie Ihre Antwort die Wiederverwendbarkeit und insbesondere die Erweiterbarkeit der Klasse beeinflusst.

   Die Klasse `GenericTableAlgorithm` kann erheblich verbessert werden, da sie momentan zwei Aufgaben erledigt. Sie würde genau wie Menschen, die in Stress geraten, wenn sie zwei Aufträge auszuführen haben und damit in den Zwiespalt konkurrierender Zuständigkeiten geraten, davon profitieren, sich nur auf eine Sache konzentrieren zu müssen.

   Die ursprüngliche Version bürdet `GenericTableAlgorithm` zwei unterschiedliche und nicht zueinander in Beziehung stehende Verantwortungen auf, die erfolgreich getrennt werden können, da sie zu völlig verschiedenen Zuständigkeitsbereichen gehören:

   – Anwendender Code BENUTZT den (entsprechend spezialisierten) generischen Algorithmus.

   – `GenericTableAlgorithm` BENUTZT die spezialisierte konkrete »Details«-Klasse, um die eigenen Operationen an einen bestimmten Fall anzupassen.

☑ Richtlinie

Setzen Sie auf Kohäsion. Bemühen Sie sich immer, jedem Codeteil – jedem Modul, jeder Klasse und jeder Funktion – eine einzige, wohl definierte Aufgabe zu geben.

Lassen Sie uns damit nun den folgenden, verbesserten Code betrachten:

```
//-----------------------------------------------------
// Datei gta.h
//-----------------------------------------------------
// Zuständigkeit #1: Bereitstellung einer öffentlichen
// Schnittstelle, die gemeinsame Funktionalität als
// Template-Methode kapselt. Dies hat mit Vererbungs-
// beziehungen nichts zu tun und könnte isoliert
// werden, um in einer besser konzentrierten Klasse
```

```
// allein stehend verwendet zu werden. Zielgruppe sind
// die externen Benutzer von GenericTableAlgorithm.
//
class GTAClient;

class GenericTableAlgorithm
{
public:
  // Der Konstructor übernimmt nun ein konkretes
  // Implementierungsobjekt.
  //
  GenericTableAlgorithm( const string& table,
                         GTAClient&    worker );

  // Da wir die Vererbungsbeziehungen entfernt haben,
  // braucht der Destruktor nicht mehr virtual zu sein.
  //
  ~GenericTableAlgorithm();

  bool Process(); // unverändert

private:
  class GenericTableAlgorithmImpl* pimpl_; // MYOB
};

//-----------------------------------------------------
// Datei gtaclient.h
//-----------------------------------------------------
// Zuständigkeit #2: Bereitstellung einer abstrakten
// Schnittstelle zur Erweiterung. Dies ist ein
// Implementierungsdetail von GenericTableAlgorithm
// und hat mit den externen Anwendern nichts zu tun.
// Es kann daher abgetrennt und in eine besser
// konzentrierte abstrakte Protokollklasse platziert
// werden. Zielgruppe sind die Autoren konkreter
// "Implementierungsdetail"-Klassen, die
// GenericTableAlgorithm benutzen (und erweitern).
//
class GTAClient
{
public:
  virtual ~GTAClient() =0;
  virtual bool Filter( const Record& );
  virtual bool ProcessRow( const PrimaryKey& ) =0;
};

//-----------------------------------------------------
// Datei gtaclient.cpp
//-----------------------------------------------------
bool GTAClient::Filter( const Record& )
```

```
{
  return true;
}
```

Diese zwei Klassen sollten so wie gezeigt in zwei getrennten Header-Dateien erscheinen. Wie wirken sich nun diese Änderungen auf den anwendenden Code aus? Die Antwort: fast gar nicht.

```
class MyWorker : public GTAClient
{
  // ... überschreibe Filter() und ProcessRow() zur
  //     Implementierung einer spezifischen Operation ...
};

int main()
{
  GenericTableAlgorithm a( "Kunde", MyWorker() );
  a.Process();
}
```

Das sieht zwar ziemlich identisch aus, hat aber drei wichtige Auswirkungen.

1. Was passiert, wenn die öffentliche Schnittstelle von `GenericTableAlgorithm` geändert wird (zum Beispiel durch Hinzufügen eines neuen öffentlichen Elementes)? In der ursprünglichen Version hätten alle konkreten Arbeiterklassen neu übersetzt werden müssen, da sie von `GenericTableAlgorithm` abgeleitet sind. In dieser Version wird jede Änderung der öffentlichen Schnittstelle von `GenericTableAlgorithm` isoliert, so dass die konkreten Arbeiterklassen davon nicht betroffen sind.

2. Was passiert, wenn das erweiterbare Protokoll von `GenericTableAlgorithm` geändert wird (zum Beispiel durch Hinzufügen zusätzlicher Default-Argumente zu `Filter()` und `ProcessRow()`)? Obwohl die öffentliche Schnittstelle dadurch unverändert bleibt, hätten in der ursprünglichen Version alle externen Anwendungen von `GenericTableAlgorithm` neu übersetzt werden müssen, da die Ableitungsschnittstelle in der Klassendefinition sichtbar ist. In dieser Version wird jede Änderung des Erweiterungsprotokolls von `GenericTableAlgorithm` isoliert, so dass die externen Anwendungen davon nicht betroffen sind.

3. Jede konkrete Arbeiterklasse kann jetzt innerhalb jedes anderen Algorithmus' benutzt werden, der über die `Filter()`/`ProcessRow()`-Schnittstelle operiert, und nicht nur in `GenericTableAlgorithm`. Dieses Resultat ist in der Tat dem Strategiemuster (*strategy pattern*) sehr ähnlich.

   Erinnern Sie sich an das Motto der Informatik: Die meisten Probleme können dadurch gelöst werden, dass eine weitere Indirektheit hinzugefügt wird. Allerdings sollte man die Dinge natürlich auch nicht komplexer als nötig machen. Die richtige Balance erhöht in diesem Fall die Wiederverwendbarkeit und Wartbarkeit erheblich, und das bei geringem oder gar keinem zusätzlichen Aufwand. Nach all den bisherigen Ausführungen ist das doch ein gutes Geschäft.

Lassen Sie uns noch einen Moment bei der »Generik« verweilen. Sie haben eventuell schon gemerkt, dass `GenericTableAlgorithm` anstatt einer Klasse sogar eine Funktion sein könnte (tatsächlich sind einige Leute geneigt, `Process()` in `operator()()` umzubenennen, da die Klasse dann offensichtlich wirklich nur ein Funktor ist). Der Grund dafür, warum die Klasse durch eine Funktion ersetzt werden könnte, ist der, dass es aufgrund der Spezifikation nicht nötig ist, zwischen den `Process()`-Aufrufen Statusinformationen zu speichern. Folgender Ersatz wäre zum Beispiel denkbar:

```
bool GenericTableAlgorithm(
  const string& table,
  GTAClient&    method
  )
{
  // ... hier: ursprünglicher Inhalt von Process() ...
}

int main()
{
  GenericTableAlgorithm( "Kunde", MyWorker() );
}
```

Was wir hier haben, ist eine generische Funktion, die je nach Bedarf mit »spezialisiertem« Verhalten ausgestattet werden kann. Wenn man von den `method`-Objekten weiß, dass sie keine Statusinformationen zu speichern brauchen (also alle Instanzen funktionell äquivalent sind und nur virtuelle Funktionen aufweisen), kann man sich das Leben einfach und `method` zu einem Nichtklassen-Template-Parameter machen.

```
template<typename GTACworker>
bool GenericTableAlgorithm( const string& table )
{
  // ... hier: ursprünglicher Inhalt von Process() ...
}

int main()
{
  GenericTableAlgorithm<MyWorker>( "Kunde" );
}
```

Ich denke nicht, dass das hier viel bringt, außer dass im anwendenden Code ein Komma eingespart wird. Die erste Funktion ist daher besser. Man sollte nicht der Versuchung erliegen, cleveren Code nur um seiner selbst willen zu schreiben.

Ob in einer gegebenen Situation eine Funktion oder eine Klasse zu benutzen ist, hängt davon ab, was man erreichen will. In diesem Fall stellt eine generische Funktion möglicherweise die bessere Lösung dar.

Lektion 24: Gebrauch und Missbrauch von Vererbung Schwierigkeitsgrad: 6

»Warum vererben?«

Vererbung wird sogar von erfahrenen Entwicklern häufig zu oft benutzt. Minimieren Sie immer die Kopplungen. Wenn es mehrere Möglichkeiten gibt, um eine Klassenbeziehung auszudrücken, benutzen Sie immer die schwächste Beziehung, die noch verwendet werden kann. Da Vererbung annähernd die stärkste Bindung ist, die man in C++ formulieren kann (nur noch von `friend`-Beziehungen übertroffen), ist sie wirklich nur dann geeignet, wenn es keine schwächere, äquivalente Alternative gibt.

Diese Lektion richtet ihr Interesse auf private Vererbung, auf eine echte (wenn auch obskure) Verwendung von `protected`-Vererbung und auf die Rekapitulation der korrekten Benutzung der öffentlichen Vererbung. Nebenbei werden wir ein ziemlich vollständiges Verzeichniss aller üblichen und aller ungewöhnlichen Gründe für den Einsatz von Vererbung zusammenstellen.

Das folgende Template stellt Funktionen zur Listenverwaltung zur Verfügung, einschließlich der Fähigkeit, Listenelemente an spezifischen Positionen innerhalb der Liste zu manipulieren.

```
// Beispiel 1
//
template <class T>
class MyList
{
public:
  bool   Insert( const T&, size_t index );
  T      Access( size_t index ) const;
  size_t Size() const;
private:
  T*     buf_;
  size_t bufsize_;
};
```

Betrachten Sie den folgenden Code, in dem zwei Möglichkeiten aufgezeigt werden, eine `MySet`-Klasse durch Verwendung von `MyList` zu schreiben. Nehmen Sie an, alle wichtigen Elemente würden gezeigt.

```
// Beispiel 1(a)
//
template <class T>
class MySet1 : private MyList<T>
{
public:
  bool   Add( const T& ); // ruft Insert() auf
  T      Get( size_t index ) const;
                          // ruft Access() auf
```

```
  using MyList<T>::Size;
  //...
};

// Beispiel 1(b)
//
template <class T>
class MySet2
{
public:
  bool   Add( const T& ); // ruft impl_.Insert() auf
  T      Get( size_t index ) const;
                          // ruft impl_.Access() auf
  size_t Size() const;    // ruft impl_.Size(); auf
  //...
private:
  MyList<T> impl_;
};
```

Denken Sie über diese Alternativen ein wenig nach und betrachten Sie dann die folgenden Fragen.

1. Gibt es zwischen `MySet1` und `MySet2` irgendwelche Unterschiede?
2. Worin besteht allgemein gesagt der Unterschied zwischen nichtöffentlicher Vererbung und Komposition? Zählen Sie so umfassend wie möglich alle Gründe auf, die Ihrer Meinung nach für Vererbung anstatt Komposition sprechen.
3. Welche Version würden Sie bevorzugen – `MySet1` oder `MySet2`?
4. Zählen Sie zum Schluss so umfassend wie möglich alle Gründe auf, die Ihrer Meinung nach für öffentliche Vererbung sprechen.

Lösung

Mit Hilfe dieses Beispiels lassen sich einige Aspekte rund um die Themen Vererbung und insbesondere nichtöffentliche Vererbung kontra Komposition illustrieren.

Frage 1 – ob es zwischen `MySet1` und `MySet2` irgendwelche Unterschiede gibt – ist leicht zu beantworten: Es gibt keine wesentlichen Unterschiede. Sie sind funktionell identisch.

Frage 2 lässt uns gleich zur Sache kommen: Worin besteht allgemein gesagt der Unterschied zwischen nichtöffentlicher Vererbung und Komposition als Elementvariable? Zählen Sie so umfassend wie möglich alle Gründe auf, die Ihrer Meinung nach für Vererbung anstatt Komposition sprechen.

- *Nichtöffentliche Vererbung* sollte immer als IST-IMPLEMENTIERT-MIT ausgedrückt werden (mit einer einzigen, seltenen Ausnahme, zu der wir gleich kommen). Die

benutzende Klasse wird dadurch von den `public`- und von den `protected`-Elementen der benutzten Klasse abhängig.

- *Komposition* drückt immer HAT-EIN aus, und folglich auch IST-IMPLEMENTIERT-MIT. Die benutzende Klasse wird dadurch nur von den `public`-Elementen der benutzten Klasse abhängig.

Man kann leicht zeigen, dass Vererbung eine Obermenge von einzelner Komposition darstellt, d.h. wir können mit einem einzelnen `MyList<T>`-Element auch nicht mehr anstellen, als wenn wir von `MyList<T>` geerbt hätten. Bei Vererbung sind wir natürlich auf nur ein `MyList<T>` (als Basisunterobjekt) begrenzt, für mehrere `MyList<T>`-Instanzen müssten wir schon Komposition benutzen.

☑ Richtlinie

Bevorzugen Sie Aggregation (»Komposition«, »Layering«, »HAT-EIN«, »Deligierung«) gegenüber Vererbung. Drücken Sie IST-IMPLEMENTIERT-MIT immer durch Aggregation und nicht durch Vererbung aus.

Nachdem wir das nun festgestellt haben, welche zusätzlichen Möglichkeiten bietet uns die Vererbung, die wir bei Komposition nicht haben? Oder anders formuliert: Warum überhaupt nichtöffentliche Vererbung verwenden? Die folgende Aufzählung liefert grob nach absteigender Bedeutung sortiert mehrere Gründe dafür. Der letzte Punkt zeigt interessanterweise eine sinnvolle (?) Anwendung der `protected`-Vererbung.

- *Wir wollen eine virtuelle Funktion überschreiben*. Das ist einer der klassischen *raisons d'être* für Vererbung.[7] Überschreiben wird häufig eingesetzt, um das Verhalten der benutzten Klasse anzupassen. Manchmal hat man aber auch keine andere Wahl, denn wenn die benutzte Klasse abstrakt ist, also mindestens eine reine virtuelle Funktion (*pure virtual function*) besitzt, die noch nicht überschrieben wurde, dann gibt es keine Alternative zum Vererben und Überschreiben, da das direkte Erzeugen einer Instanz nicht möglich ist.
- *Wir brauchen Zugriff auf ein protected-Element*. Dies trifft auf `protected`-Elementfunktionen[8] im Allgemeinen und auf `protected`-Konstruktoren im Besonderen zu.
- *Wir müssen das benutzte Objekt vor einem anderen Basisunterobjekt konstruieren oder nach einem anderen Basisunterobjekt zerstören*. Ist die geringfügig längere Objektlebenszeit entscheidend, gibt es keine Alternative zur Vererbung. Das kann zum Beispiel nötig werden, wenn die benutzte Klasse irgendeine Art von Sperrmecha-

7. Siehe Meyers98 unter dem Indexeintrag »French, gratuitous use of«.
8. Ich sage hier »Elementfunktionen«, denn Sie würden ja auch nie eine Klasse mit einer `public`- oder `protected`-Elementvariable schreiben, oder? (Unabhängig von einigen schlechten Beispielen, die man in diversen Bibliotheken findet.)

nismus betätigt (wie ein kritischer Abschnitt oder eine Datenbanktransaktion), der für die gesamte Lebenszeit eines anderen Basisunterobjekts aufrechterhalten werden soll.

- *Wir müssen eine gemeinsame virtuelle Basisklasse benutzen oder wollen die Konstruktion einer virtuellen Basisklasse überschreiben*. Der erste Teil trifft zu, wenn die benutzende Klasse von einer der virtuellen Basisklassen der benutzten Klasse abgeleitet sein muss. Wenn nicht, kann immer noch der zweite Teil zutreffen. Die am weitesten abgeleitete Klasse ist für die Initialisierung aller virtuellen Basisklassen zuständig. Wenn wir also für eine virtuelle Basisklasse einen anderen Konstruktor oder andere Konstruktorparameter verwenden wollen, dann geht das nur durch Vererbung.

- *Wir profitieren wesentlich von der Optimierung leerer Basisklassen*. Manchmal hat die Klasse, MIT der wir IMPLEMENTIEREN, keinerlei Datenelemente, stellt also nur eine Ansammlung von Funktionen dar. In diesem Fall kann es bei Vererbung im Gegensatz zur Komposition aufgrund der Optimierung leerer Basisklassen zu einem Speicherplatzvorteil kommen. Dem Compiler ist es nämlich erlaubt, ein leeres Basisunterobjekt keinen Speicherplatz verbrauchen zu lassen. Das gleiche Objekt als Elementvariable darf dagegen nicht eine Größe von null haben, auch wenn es keine Daten enthält.

```
class B { /* ... nur Funktionen, keine Daten ... */ };
// Komposition: führt zu leichtem Platz-Overhead
//
class D
{
  B b_; // b_ muss mindestens ein Byte verbrauchen,
};      // sogar wenn B eine leere Klasse ist
// Vererbung: kann zu gar keinem Platz-Overhead führen
//
class D : private B
{       // B-Basisunterobjekt braucht überhaupt
};      // keinen Speicherplatz belegen
```

Eine detaillierte Besprechung der Optimierung leerer Basisklassen finden Sie in dem entsprechenden, exzellenten Artikel von Nathan Meyers im *Dr. Dobb's Journal* (Meyers97).

Lassen Sie mich noch eine Warnung an die Übereifrigen hinzufügen: Nicht alle Compiler führen die Optimierung leerer Basisklassen durch. Aber selbst wenn, dann lohnt es sich erst dann merklich, wenn Sie viele (sagen wir Zehntausende) solcher Objekte im System haben. Solange diese Platzersparnis nicht wirklich wichtig für Ihre Anwendung ist und solange Sie nicht wissen, ob Ihr Compiler die Opitimierung wirklich durchführt, wäre es ein Fehler, die durch die Vererbung als im Gegensatz zur Komposition stärkere Beziehung verursachte Kopplung einzuführen.

Es gibt nur eine zusätzliche Eigenschaft, die wir durch nichtöffentliche Vererbung erreichen können, und diese Eigenschaft ist die einzige, durch die dabei nicht IST-IMPLEMENTIERT-MIT modelliert wird:

- *Wir brauchen »kontrollierten Polymorphismus« – LSP IST-EIN, aber nur in bestimmtem Code*. Öffentliche Vererbung sollte immer für eine IST-EIN-Beziehung nach dem Liskov-Substitutionsprinzip (LSP) stehen.[9] Obgleich die meisten Leute IST-EIN allein mit öffentlicher Vererbung identifizieren, kann man durch nichtöffentliche Vererbung eine beschränkte Form der IST-EIN-Beziehung ausdrücken. Von außen betrachtet stellt bei einer Klasse `Derived: private Base` ein `Derived`-Objekt KEIN `Base`-Objekt dar. Aufgrund der durch die private Vererbung verursachten Zugriffsbeschränkungen kann es somit natürlich auch nicht polymorph als `Base`-Objekt benutzt werden. *Innerhalb* der Elementfunktionen und der `friend`-Funktionen von `Derived` kann ein `Derived`-Objekt allerdings polymorph als `Base`-Objekt benutzt werden (man kann einen Zeiger oder eine Referenz auf ein `Derived`-Objekt an Stellen übergeben, wo `Base` erwartet wird), da diese Funktionen über die nötigen Zugriffsrechte verfügen. Ersetzt man die private Vererbung durch `protected`-Vererbung, wird die IST-EIN-Beziehung auch für entferntere abgeleitete Klassen sichtbar, so dass diese Unterklassen ebenfalls Gebrauch vom Polymorphismus machen können.

Das ist die vollständigste Aufzählung aller Gründe für nichtöffentliche Vererbung, die ich aufstellen kann. (Tatsächlich fehlt nur noch ein Punkt, der daraus eine Liste aller Gründe für jede Art von Vererbung machen würde: Um IST-EIN auszudrücken, brauchen wir öffentliche Vererbung. Mehr dazu in der Behandlung von Frage 4.)

All das führt uns zu Frage 3: Welche Version würden Sie bevorzugen – `MySet1` oder `MySet2`? Lassen Sie uns den Code in Beispiel 1 analysieren und prüfen, ob eines der obigen Kriterien zutrifft.

- `MyList` hat keine `protected`-Elemente. Um Zugriff auf die Elemente zu bekommen, ist keine Vererbung nötig.
- `MyList` hat keine virtuellen Funktionen, so dass wir nicht ableiten müssen, um diese zu überschreiben.
- `MyList` hat keine anderen potenziellen Basisklassen, so dass das `MyList`-Objekt nicht vor einem anderen Basisunterobjekt konstruiert oder nach einem anderen Basisunterobjekt zerstört werden muss.
- `MyList` hat keine virtuelle Basisklasse, die von `MySet` mitbenutzt werden könnte oder deren Konstruktion überschrieben werden müsste.

9. Mehrere gute Arbeiten über LSP finden Sie unter www.objectmentor.com.

- `MyList` ist nicht leer, so dass die Optimierung leerer Basisklassen hier nicht greifen kann.
- `MySet` IST-NICHT-EIN `MyList`, auch nicht innerhalb Element- und `friend`-Funktionen von `MySet`. Dieser letzte Aspekt ist interessant, da er einen (kleinen) Nachteil der Vererbung aufdeckt: Auch wenn eins der obigen Kriterien zutrifft, so dass wir uns für Vererbung entscheiden, müssen wir trotzdem sorgfältig darauf achten, dass die Element- und `friend`-Funktionen von `MySet` ein `MySet`-Objekt nicht aus Versehen polymorph als ein `MyList`-Objekt benutzen. Nun ist diese Möglichkeit zwar recht unwahrscheinlich, kann aber die davon betroffenen armen Programmierer stundenlang in die Irre führen.

Kurz gesagt sollte `MySet` nicht von `MyList` abgeleitet werden. Vererbung an Stellen einzusetzen, an denen Komposition genauso effektiv ist, führt nur grundlos zu Kopplungen und zu unnötigen Abhängigkeiten, und das ist nie gut. In der Realität sehe ich leider noch immer viele – selbst erfahrene – Programmierer, die Beziehungen wie bei `MySet` durch Vererbung implementieren.

Findige Leser werden bereits gemerkt haben, dass die vererbungsbasierte `MySet`-Version gegenüber der auf Komposition basierenden Version einen (ziemlich belanglosen) Vorteil aufweist: Bei Vererbung reicht eine `using`-Anweisung aus, um die unveränderte `Size()`-Funktion zugänglich zu machen. Bei Komposition bedarf es dazu schon einer expliziten, zusätzlichen Funktion.

Natürlich ist manchmal auch Vererbung das geeignetste Mittel. Zum Beispiel:

```
// Beispiel 2: Manchmal ist Vererbung nötig
//
class Base
{
public:
  virtual int Func1();
protected:
  bool Func2();
private:
  bool Func3(); // benutzt Func1()
};
```

Vererbung ist notwendig, wenn wir eine virtuelle Funktion wie `Func1()` überschreiben oder auf ein `protected`-Element zugreifen wollen. Beispiel 2 verdeutlicht, dass es auch andere Gründe für das Überschreiben virtueller Funktionen geben kann als das Ermöglichen von Polymorphismus. `Base` ist hier mit `func1()` implementiert (`func3()` benutzt `func1()` in der Implementierung), so dass das korrekte Verhalten nur durch Überschreiben von `func1` erreicht werden kann. Wenn also Vererbung notwendig ist, stellt dann der folgende Code die richtige Methode zur ihrer Realisierung dar?

```
// Beispiel 2(a)
//
class Derived : private Base // nötig?
{
public:
  int Func1();
  // ... mehr Funktionen, einige benutzen
  //     Base::Func2(), andere nicht ...
};
```

In diesem Code kann `Derived` wie gewünscht `Base::Func1()` überschreiben. Leider wird damit *allen Elementen* von `Derived` der Zugriff auf `Base::Func2()` ermöglicht, und da liegt der Hase im Pfeffer. Vielleicht brauchen nur einige oder sogar nur eine Elementfunktion von `Derived` auf `Base::Func2()` zuzugreifen. So, wie die Vererbung hier benutzt wird, macht sie alle Elemente von `Derived` von der `protected`-Schnittstelle von `Base` abhängig.

Sicher, Vererbung ist notwendig. Aber wäre es nicht schön, wenn wir nur so viel Kopplung verursachen würden, wie wir wirklich brauchen? Mit ein wenig Umsicht ist dies tatsächlich möglich.

```
// Beispiel 2(b)
//
class DerivedImpl : private Base
{
public:
  int Func1();
  // ... Funktionen, die Func2() benutzen ...
};
class Derived
{
  // ... Funktionen, die Func2() nicht benutzen ...
private:
  DerivedImpl impl_;
};
```

Dieser Entwurf ist viel besser, da er die Abhängigkeiten von `Base` abtrennt und isoliert. `Derived` hängt jetzt nur noch direkt von den öffentlichen Schnittstellen von `Base` und `DerivedImpl` ab. Warum ist dieses Design erfolgreicher? Hauptsächlich deshalb, weil es sich an das fundamentale Entwurfsprinzip »eine Klasse, eine Verantwortlichkeit« hält. In Beispiel 2(a) ist `Derived` sowohl für die Anpassung von `Base` als auch für die eigene Implementierung mit `Base` zuständig. In Beispiel 2(b) sind diese beiden Aufgaben genau aufgeteilt.

Betrachten wir nun einige Varianten bei der Komposition. Komposition hat ihre eigenen Vorteile. Erstens erlaubt sie mehrere Instanzen der benutzten Klasse, was bei Vererbung unpraktisch oder sogar unmöglich ist. Verwenden Sie das gleiche Idiom wie in Beispiel 2(b), falls Sie sowohl Ableitung als auch mehrere Instanzen benötigen. Leiten

Sie eine Hilfsklasse (wie `DerivedImpl`) ab, um die Vererbung zu nutzen, und setzen Sie dann mehrere Instanzen dieser Hilfsklasse ein.

Zweitens bringt es mehr Flexibilität, wenn die benutzte Klasse als Datenelement verwendet wird. Das Element kann innerhalb eines Pimpl[10] hinter einem Compiler-Firewall verborgen werden (wohingegen Basisklassendefinitionen stets sichtbar sein müssen), und es kann leicht in einen Zeiger umgewandelt werden, wenn es zur Laufzeit verändert werden soll (während Vererbungshierarchien statisch sind und zur Übersetzungszeit feststehen).

Zum Schluss hier noch eine dritte Möglichkeit, `MySet2` aus Beispiel 1(b) so umzuschreiben, dass Komposition in allgemeinerer Form eingesetzt wird:

```
// Beispiel 1(c): generische Komposition
//
template <class T, class Impl = MyList<T> >
class MySet3
{
public:
  bool   Add( const T& ); // ruft impl_.Insert() auf
  T      Get( size_t index ) const;
                          // ruft impl_.Access() auf
  size_t Size() const;    // ruft impl_.Size(); auf
  // ...
private:
  Impl impl_;
};
```

`MySet` ist jetzt nicht mehr nur IMPLEMENTIERT MIT `MyList<T>`, sondern MIT jeder Klasse *IMPLEMENTIERBAR*, die `Add()`, `Get()` und die anderen benötigten Funktionen unterstützt. Die C++-Standardbibliothek benutzt genau diese Technik für ihre `stack`- und `queue`-Templates, die voreingestellt MIT einer `deque` IMPLEMENTIERT sind, aber genauso gut MIT jeder anderen Klasse *IMPLEMENTIERBAR* sind, die die erforderlichen Dienste unterstützt.

Insbesondere ist es dem Anwender möglich, die Implementierung für die `MySet`-Instantiierung nach der Performance auszusuchen. Wenn wir zum Beispiel wissen, dass wir viel öfter einfügen werden als suchen, dann benutzen wir eine Implementierung, die für Einfügen optimiert ist. Dabei ist die Anwendung in keinster Weise komplizierter geworden. In Beispiel 1(b) muss man einfach `MySet2<int>` schreiben, um einen Satz `int`s zu erzeugen. Für Beispiel 1(c) trifft das immer noch zu, denn `MySet3<int>` ist dank der Template-Default-Parameter nur ein Synonym für `MySet3<int,MyList<int> >`.

10. Siehe Lektionen 26 bis 30.

Eine solche Flexibilität mit Vererbung zu erreichen ist im Wesentlichen deshalb viel schwieriger, weil Design-Entscheidungen beim Einsatz von Vererbung meistens schon zur Entwurfszeit engültigen Charakter haben.

Man kann Beispiel 1(c) so umschreiben, dass von `Impl` abgeleitet wird, aber die dadurch verursachte stärkere Kopplung ist unnötig und sollte vermieden werden.

Das Wichtigste über öffentliche Vererbung kann man durch die Antwort auf Frage 4 lernen: Zählen Sie so umfassend wie möglich alle Gründe auf, die Ihrer Meinung nach für öffentliche Vererbung sprechen.

Bezüglich der öffentlichen Vererbung werde ich mich nur auf einen Aspekt konzentrieren, und wenn Sie diesem Ratschlag folgen, werden Sie die meisten oft gemachten Fehler und Missbrauchsfälle umschiffen. Setzen Sie öffentliche Vererbung *nur dann* ein, wenn Sie eine wirkliche IST-EIN-Beziehung per Liskov-Substitutionsprinzip[11] modellieren wollen. Das heißt, eine Instanz einer abgeleiteten Klasse muss überall dort und in jedem Kontext einsetzbar sein, wo auch eine Instanz der Basisklasse benutzbar ist. Die Bedeutungen müssen dabei erhalten bleiben. (Hinweis: Eine seltene Ausnahme, oder besser Erweiterung, dieses Prinzips haben wir in Lektion 3 kennen gelernt.)

Besonders zwei Fallstricke vermeidet man durch Befolgen dieser Regel.

- *Benutzen Sie niemals öffentliche Vererbung, wenn nichtöffentliche ausreichend ist*. Mit öffentlicher Vererbung sollte man keine IST-IMPLEMENTIERT-MIT-Beziehung modellieren, wenn keine wirkliche IST-EIN-Beziehung vorhanden ist. Das scheint offensichtlich zu sein, aber ich habe bemerkt, dass einige Programmierer aus Gewohnheit die Vererbung öffentlich machen. Das ist keine gute Idee. Im gleichen Geist sehe ich meinen Ratschlag, niemals Vererbung zu benutzen (egal welcher Art), wenn die gute alte Komposition ausreichend ist. Wenn die zusätzlichen Zugriffsmöglichkeiten und die stärkere Kopplung nicht nötig sind, warum sie dann benutzen? Wenn es mehrere Wege gibt, eine Klassenbeziehung auszudrücken, sollte man immer den schwächstmöglichen benutzen.
- *Benutzen Sie niemals öffentliche Vererbung, um »IST-FAST-IMMER-EIN« zu implementieren*. Ich habe Programmierer gesehen, sogar erfahrene, die öffentlich von einer Basisklasse abgeleitet und die »meisten« der überschriebenen virtuellen Funktionen so implementiert haben, dass die Semantiken der Basisklasse erhalten blieben. Anders formuliert gibt es dadurch Fälle, in denen sich ein `Derived`-Objekt an Stelle eines `Base`-Objekts nicht so verhält, wie das ein Anwender von `Base` erwarten würde. Ein oft zitiertes Beispiel von Robert Martin ist die gewöhnlich irrige Idee, eine `Square`-Klasse von einer `Rectangle`-Klasse abzuleiten, nur »weil ein Quadrat ein Rechteck ist«. Das mag ja in der Mathematik wahr sein, ist es aber nicht notwendigerweise bei Klassen. Nehmen wir zum Beispiel an, die Klasse `Rectangle` habe eine

11. Mehrere gute Arbeiten über LSP finden Sie unter www.objectmentor.com.

virtuelle Funktion `SetWidth(int)`. Deren Implementierung bei `Square` würde natürlich auch die Höhe verändern, damit das Objekt quadratisch bleibt. Irgendwo im System könnte es nun Code geben, der polymorph mit `Rectangle`-Objekten arbeitet und nicht erwartet, dass eine Veränderung der Breite auch die Höhe beeinflusst. Im Allgemeinen ist das bei Rechtecken ja auch nicht der Fall. Dies ist eine gutes Beispiel für öffentliche Vererbung, die das LSP verletzt, weil die abgeleitete Klasse nicht die gleichen Bedeutungen wie die Basisklasse hat. Die Leitlinie der öffentlichen Vererbung wird missachtet: »Fordere nicht mehr und versprich nicht weniger.«

Den Leuten, bei denen ich diese Art des »IST-FAST-EIN« sehe, versuche ich klarzumachen, welchen Ärger sie sich damit einhandeln. Schließlich wird irgendjemand irgendwo zwangsläufig versuchen, die abgeleiteten Objekte in einer der Arten und Weisen polymorph einzusetzen, die gelegentlich unerwartete Resultate liefern, oder? »Aber das ist in Ordnung«, bekam ich einmal als Antwort, »es handelt sich nur um eine kleine Inkompatibilität, und ich weiß, dass niemand die Objekte der `Base`-Familie in dieser Weise benutzen wird [Das wäre gefährlich].« Tja, »ein wenig inkompatibel« ist das Gleiche wie »ein wenig schwanger«. Im Moment gibt es keinen Grund, an dem Programmierer zu zweifeln – also daran, dass im System kein Code von den gefährlichen Unterschieden betroffen ist. Aber ebenso wenig zweifle ich daran, dass eines Tages, irgendwo, ein Wartungsprogrammierer eine scheinbar harmlose Änderung vornehmen und damit einen Haufen Probleme lostreten wird. Stundenlang wird er analysieren, warum das Klassendesign so schlecht ist, und weitere Tage damit zubringen, den Fehler zu beheben.

Erliegen Sie nicht der Versuchung. Sagen Sie einfach Nein. Wenn es sich nicht wie `Base` verhält, ist es NICHT `Base`, also erzeugen Sie keine Ableitungen, die so aussehen, als wären sie es.

☑ Richtlinie

Stellen Sie stets sicher, dass öffentliche Vererbung sowohl IST-EIN als auch FUNKTIONIERT-WIE-EIN nach dem Liskov-Substitutionsprinzip modelliert. Alle überschriebenen Elementfunktionen dürfen nicht mehr erfordern und nicht mehr versprechen.

☑ Richtlinie

Erben Sie niemals öffentlich, um Code wiederzuverwenden (aus der Basisklasse). Erben Sie nur öffentlich, um selbst wiederverwendet zu werden (von Code, der die Basisobjekte polymorph benutzt).

4.1 Schlussbemerkung

Setzen Sie Vererbung weise ein. Wenn Sie die Möglichkeit haben, eine Klassenbeziehung allein durch Komposition/Delegierung auszudrücken, sollten Sie auch davon Gebrauch machen. Verwenden Sie nichtöffentliche Vererbung, wenn Sie zwar Vererbung brauchen, aber keine IST-EIN-Beziehung modellieren. Wenn Sie die Mächtigkeit der Mehrfachvererbung nicht brauchen, bevorzugen Sie einfache Vererbung. Große Vererbungshierarchien (insbesondere sehr tiefe) sind verwirrend, schlecht verständlich und daher schwierig zu warten. Vererbung ist eine Entscheidung zur Entwurfszeit, für die man eine Menge Laufzeitflexibilität einbüßt.

Einige Leute meinen, »dass es ohne Vererbung eben nicht objektorientiert ist«, aber das ist nicht ganz richtig. Benutzen Sie die einfachste Lösung, die funktioniert, und Sie werden sich viel eher jahrelang an stabilem und wartbarem Code erfreuen können.

Lektion 25: Objektorientierte Programmierung **Schwierigkeitsgrad: 4**

Ist C++ eine objektorientierte Sprache? Ja und, im Gegensatz zur üblichen Ansicht, auch nein. Hier zur Abwechslung (und um für einen Moment von der Codeanalyse wegzukommen) eine essayistische Frage.

Denken Sie ein wenig darüber nach und wechseln Sie nicht gleich zur Antwort.

Nehmen Sie zur folgenden Aussage Stellung: »C++ ist eine mächtige Sprache, die über viele hochentwickelte objektorientierte Charakteristika verfügt, einschließlich Kapselung, Exception-Handling, Vererbung, Templates, Polymorphismus, eines strengen Typsystems und eines vollständigen Modulsystems.«

Lösung

Der Zweck dieser Lektion besteht darin, Nachdenken über Hauptmerkmale (und auch über fehlende) von C++ zu provozieren und eine gesunde Dosis Realitätsbewusstsein zu verabreichen. Insbesondere hoffe ich, dass Ihnen bei der Beschäftigung mit dieser Lektion Ideen gekommen und Beispiele eingefallen sind, die drei wesentliche Dinge verdeutlichen.

1. *Nicht jeder versteht unter OO das Gleiche*. Aber was ist denn nun Objektorientierung? Auch heutzutage ist es noch so, dass, wenn Sie 10 Leute danach fragen, Sie höchstwahrscheinlich 15 verschiedene Antworten erhalten. (Das ist auch nicht besser als 10 Anwälte um Rechtsauskünfte zu bitten.)

   Praktisch fast jeder wird zustimmen, dass Vererbung und Polymorphismus OO-Konzepte sind. Die meisten Leute zählen auch Kapselung dazu. Einige würden noch Exception-Handling hinzufügen und vielleicht niemand Templates. Es gibt

eben verschiedene Ansichten darüber, ob ein bestimmtes Merkmal ein OO-Merkmal darstellt, und jeder Standpunkt hat seine leidenschaftlichen Verteidiger.

2. *C++ ist eine Multiparadigmasprache*. C++ ist nicht nur eine OO-Sprache. Es hat viele OO-Eigenschaften, zwingt den Programmierer aber nicht dazu, diese auch zu benutzen. Man kann vollständige Nicht-OO-Programme in C++ schreiben, was auch von vielen Leuten gemacht wird.

   Der wichtigste Beitrag der C++-Standardisierung zu C++ ist die verbesserte Unterstützung starker Abstraktion zur Reduzierung der Software-Komplexität (Martin95).[12] C++ ist nicht ausschließlich eine objektorientierte Sprache. Es ermöglicht mehrere Programmierstile, einschließlich objektorientierter und generischer Programmierung. Diesen kommt durch ihre Eigenschaft, flexible Methoden zur Codeorganisation durch Abstraktion zur Verfügung zu stellen, eine fundamentale Bedeutung zu. Mit objektorientierter Programmierung können wir den Zustand eines Objektes mit den es manipulierenden Funktionen verbinden, und Einkapselung und Vererbung ermöglichen uns die Verwaltung der Abhängigkeiten und lassen eine sauberere und leichterere Wiederbenutzung zu. Generische Programmierung ist ein jüngerer Stil, mit dem wir Funktionen und Klassen schreiben können, die Operationen auf andere Funktionen und Objekte nicht näher benannten, nicht in Zusammenhang stehenden und unbekannten Typs anwenden. Diese Vorgehensweise stellt eine einzigartige Methode dar, um die Kopplungen und gegenseitigen Abhängigkeiten innerhalb eines Programms zu reduzieren. Generische Programmierung ist auch in einigen anderen Sprachen möglich, aber keine unterstützt dies so stark wie C++. Tatsächlich wurde die moderne generische Programmierung durch die einmalige C++-Formulierung der Templates ermöglicht.

   Heutzutage stehen in C++ viele mächtige Methoden zur Verfügung, um Abstraktion auszudrücken. Die dadurch entstandene Flexibilität ist das wichtigste Ergebnis der C++-Standardisierung.

3. *Keine Sprache ist das ultimative Universalwerkzeug*. Heute benutze ich C++ als meine Hauptprogrammiersprache; morgen werde ich das benutzen, was für mein Vorhaben am besten geeignet ist. C++ hat kein Modulsystem (vollständig oder wie auch immer); es fehlen ihm andere wichtige Merkmale wie zum Beispiel Garbage-Collection; und es hat ein statisches, jedoch nicht notwendigerweise »robustes« Typsystem. Jede Sprache hat ihre Vor- und Nachteile. Wählen Sie einfach das richtige Werkzeug für Ihr Problem und erliegen Sie nicht der Versuchung, ein kurzsichtiger Sprachenfanatiker zu werden.

12. Das Buch enthält eine exzellente Erörterung der Frage, warum einer der wichtigsten Vorteile objektorientierter Programmierung darin besteht, die Software-Komplexität durch Verwaltung der Codeabhängigkeiten zu reduzieren.

5 Compiler-Firewalls und das Pimpl-Idiom

Abhängigkeiten erfolgreich zu managen ist einer der wesentlichen Punkte beim Schreiben von solidem Code. Ich habe behauptet (Sutter98), dass die größte Stärke von C++ die Unterstützung zweier mächtiger Abstraktionsmethoden ist, der objektorientierten Programmierung und der generischen Programmierung. Beide sind grundlegende Werkzeuge für den Umgang mit Abhängigkeiten und somit bei der Handhabung von Komplexität. Es spricht für sich, dass all die verbreiteten Schlüsselwörter der OOP und generischen Programmierung – einschließlich solcher wie Kapselung, Polymorphismus und Typunabhängigkeit – und weiterhin auch alle Entwurfsmuster, die ich kenne, Wege beschreiben, um der Komplexität in einem Softwaresystem Herr zu werden, indem die gegenseitigen Abhängigkeiten im Code geschickt organisiert werden.

Lektion 26: Minimieren der Übersetzungszeit-Abhängigkeiten – Teil 1 — Schwierigkeitsgrad: 4

Wenn wir über Abhängigkeiten sprechen, denken wir gewöhnlich an Laufzeit-Abhängigkeiten wie Interaktionen von Klassen. In dieser Lektion werden wir unser Augenmerk stattdessen darauf richten, Übersetzungszeit-Abhängigkeiten zu analysieren und zu managen. Versuchen Sie als ersten Schritt unnötige Headerdateien zu identifizieren (und zu beseitigen).

Viele Programmierer binden aus Gewohnheit via `#include` viel mehr Headerdateien als nötig ein. Leider kann diese Vorgehensweise zu erheblichen Verlängerungen der Übersetzungszeit führen, insbesondere, wenn eine beliebte Headerdatei viel zu viele weitere Headerdateien einbindet.

Welche `#include`-Direktiven könnten in der folgenden Headerdatei sofort entfernt werden, ohne die Funktionalität zu beeinträchtigen? Nehmen Sie keine Änderungen außer dem Entfernen oder Neuschreiben von `#include`-Direktiven vor. Beachten Sie die Informationen, die Sie den Kommentaren entnehmen können.

```
//  x.h: Original-Headerdatei
//
#include <iostream>
```

```
#include <ostream>
#include <list>
using namespace std;
// Weder A, B, C, D oder E sind Templates.
// Nur A und C haben virtuelle Funktionen.
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
#include "e.h"  // class E
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  std::list<C> clist_;
  D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Lösung

Eine der ersten beiden Standard-Headerdateien, die in `x.h` aufgeführt sind, kann sofort entfernt werden, da sie nicht benötigt wird. Die zweite Headerdatei kann durch eine kleinere Headerdatei ersetzt werden.

1. `iostream` entfernen.

   ```
   #include <iostream>
   ```

   Viele Programmierer schreiben `#include <iostream>` aus reiner Gewohnheit, sobald sie etwas sehen, was einem Stream auch nur ähnlich sieht. Die Klasse `X` benutzt Streams, das ist richtig, es wird aber nichts Besonderes aus `iostream` verwendet. Bestenfalls benötigt `X` nur `ostream`, aber selbst das kann noch eingeschränkt werden.

☑ Richtlinie

Binden Sie keine unnötigen Headerdateien ein.

2. `ostream` durch `iosfwd` ersetzen.

   ```
   #include <ostream>
   ```

   Parameter und Rückgabewerte müssen nur vorwärts-deklariert werden. Statt der vollständigen Definition von `ostream` benötigen wir also nur die entsprechende Vorwärts-Deklaration.

Vor einigen Jahren konnte man in solchen Fällen »`#include <ostream.h>`« einfach durch »`class ostream;`« ersetzen, da `ostream` eine Klasse war und sich nicht im Namensraum `std` befand. Dies ist leider nicht mehr so. Es gibt zwei Gründe, warum »`class ostream;`« nicht mehr gültig ist.

1. `ostream` befindet sich nun im Namensraum `std` und Programmierern ist nicht erlaubt irgendetwas im Namensraum `std` zu deklarieren.

2. `ostream` ist jetzt ein `typedef` für ein Template; um genau zu sein ist es ein `typedef` für `basic_ostream<char>`. Eine Vorwärts-Deklaration von `basic_ostream` wäre nicht nur unschön, sie wäre gar nicht immer möglich, da Implementierungen von Standardbibliotheken solche Dinge erlaubt sind, wie zusätzliche Template-Parameter (neben dem vom Standard vorgeschriebenen) einzuführen, von denen Ihr Code natürlich nichts wissen kann – einer der wichtigsten Gründe für die Regel, dass Programmierer im Namensraum `std` nichts deklarieren dürfen.

   Es ist jedoch nicht alles verloren. Die Standardbibliothek stellt dankenswerterweise die Headerdatei `iosfwd` zur Verfügung, die Vorwärtsdeklarationen für alle Stream-Templates (einschließlich `basic_ostream`) und für alle Standard-`typedef`s (einschließlich `ostream`) enthält. Somit ist alles, was wir tun müssen, »`#include <ostream>`« mit »`#include <iosfwd>`« zu ersetzen.

☑ Richtlinie

Bevorzugen Sie `#include <iosfwd>`, wenn eine Vorwärts-Deklaration für einen Stream ausreicht.

Nebenbei bemerkt, wenn man `iosfwd` zum ersten Mal sieht, könnte man denken, dass der gleiche Trick auch für andere Standard-Templates wie `list` oder `string` funktioniert. Es gibt jedoch keine vergleichbaren Standard-Headerdateien wie »`stringfwd`« oder »`listfwd`«. Die `isofwd`-Headerdatei wurde aus Gründen der Rückwärtskompatibilität für Streams eingeführt, so dass alter Code, der in den Jahren der »alten« template-losen iostream-Bibliothek geschrieben wurde, nicht unbrauchbar gemacht wird.

So, das war einfach. Wir können ...

»Was? Nicht so schnell!«, kann ich einige von Ihnen rufen hören. »Diese Headerdatei macht viel mehr mit `ostream`, als `ostream` nur als Parameter oder Rückgabewert aufzuführen. Der `inline operator<<` benutzt ein ostream-Objekt! Dafür wird doch die Definition von `ostream` benötigt, oder?«

Das ist eine gute Frage. Zum Glück ist die Antwort: Nein, sie wird nicht benötigt. Betrachten Sie den fraglichen Operator noch mal:

```
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Diese Funktion führt `ostream&` als Parameter und Rückgabewert auf (was beides keine Definition von `ostream` erfordert, wie die meisten Leute wissen) und gibt den eigenen `ostream&`-Parameter als Parameter an eine andere Funktion weiter (was auch keine Definition von `ostream` erfordert, wie die meisten Leute *nicht* wissen). Solange dies alles ist, was wir mit dem `ostream&`-Parameter machen, besteht keine Notwendigkeit für die vollständige Definition von `ostream`. Natürlich bräuchten wir diese Definition, wenn wir zum Beispiel versuchen würden, irgendeine Elementfunktion von `ostream` aufzurufen, aber wir machen ja nichts Vergleichbares hier.

Was die anderen Headerdateien betrifft, können wir uns nur einer weiteren entledigen.

3. `e.h` durch eine Vorwärtsdeklaration ersetzen.

   ```
   #include "e.h"  // class E
   ```

   Die Klasse E wird nur als Parameter und als Rückgabewert aufgeführt, so dass keine Definition erforderlich ist und `x.h` somit `e.h` auch nicht einbinden sollte. Alles, was wir machen müssen, ist, »`#include "e.h"`« mit »`class E;`« zu ersetzen.

☑ Richtlinie

Binden Sie keine Headerdatei ein, wenn eine Vorwärts-Deklaration ausreicht.

Lektion 27: Minimieren der Übersetzungszeit-Abhängigkeiten – Teil 2 Schwierigkeitsgrad: 6

Nun, da wir die unnötigen Headerdateien entfernt haben, können wir mit Phase 2 beginnen: Wie können Sie die Abhängigkeiten von den Interna einer Klasse minimieren?

Unten sehen Sie die Headerdatei aus Lektion 26 nach der ersten Optimierungsphase. Welche weiteren `#include`-Direktiven könnten entfernt werden, wenn wir einige passende Änderungen vornehmen würden, und wie müssten diese aussehen?

Dieses Mal dürfen Sie Änderungen an der Klasse `X` vornehmen, wobei die Basisklassen von `X` und die öffentliche Schnittstelle unangetastet bleiben sollen; jeglicher Code, der `X` bereits benutzt, sollte durch die Änderungen nicht weiter betroffen sein, außer dass eine Neukompilierung nötig ist.

```
//  x.h: ohne überflüssige Headerdateien
//
#include <iosfwd>
#include <list>
// Weder A, B, C, D oder E sind Templates.
// Nur A und C haben virtuelle Funktionen.
#include "a.h"  // class A
#include "b.h"  // class B
#include "c.h"  // class C
#include "d.h"  // class D
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  std::list<C> clist_;
  D            d_;
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Lösung

Es gab einige Dinge, die wir in der vorigen Lektion nicht tun konnten.

- Wir mussten `a.h` und `b.h` beibehalten. Wir könnten sie nicht rausschmeißen, da `X` sowohl von `A` als auch von `B` erbt und Sie immer die vollständige Definition aller Basisklassen benötigen, damit der Compiler die Größe von `X`-Objekten, virtuelle Funktionen und andere grundlegende Sachen bestimmen kann. (Können Sie erraten, wie eine dieser Headerdateien entfernt werden kann? Denken Sie darüber nach: Welche können Sie entfernen? Warum und wie? Die Auflösung gibt's gleich.)
- Wir mussten `list`, `c.h` und `d.h` belassen. Wir konnten sie nicht entfernen, da `list<C>` und `D` zur Definition privater Datenelemente dienen. Trotzdem `C` weder als Basisklasse noch als Datenelement auftaucht, wird die Klasse zum Instanziieren des

`list`-Datenelements verwendet und die meisten aktuellen Compiler erfordern, dass ihnen die Definition von `C` zugänglich ist, wenn Sie `list<C>` instanziieren. (Der Standard erfordert dies jedoch nicht, so dass, selbst wenn Ihr Compiler Ihnen diese Restriktion jetzt auferlegt, Sie erwarten können, dass sie mit der Zeit verschwinden wird.)

Lassen Sie uns jetzt über die Schönheit von Pimpls[1] reden. (Ja, wirklich.)

C++ lässt uns sehr einfach private Teile einer Klasse vor unautorisiertem *Zugriff* kapseln. Leider bedarf es durch das von C übernommene Einbinden von Headerdateien einiger Mehrarbeit, um die *Abhängigkeiten* der privaten Teile einer Klasse zu kapseln. »Aber«, sagen Sie, »der ganze Sinn von Kapselung ist doch, dass Client-Code nichts über die privaten Implementierungsdetails einer Klasse wissen noch sich darum kümmern muss, richtig?« Richtig, und in C++ braucht der Client-Code auch nichts über den Zugriff auf private Teile der Klasse wissen noch sich darum kümmern (weil, wenn es sich nicht gerade um einen `friend` handelt, dies sowieso nicht erlaubt ist). Da jedoch auch der private Teil einer Klasse in der zugehörigen Headerdatei sichtbar ist, hängt der Client-Code auch von allen Typen ab, die dort aufgeführt sind.

Wie können wir Clients besser von den privaten Implementierungsdetails einer Klasse isolieren? Ein gute Möglichkeit dafür ist eine spezielle Form des Handle-Body-Idioms (Coplien92) (das ich das Pimpl-Idiom nenne, wegen der möglichen Aussprache des `pimpl_`-Pointers[2]) als eine Compiler-Firewall (Lakos96, Meyers98, Meyers99, Murray93).

Ein Pimpl ist nichts weiter als ein undurchsichtiger Zeiger (ein Zeiger auf eine vorwärts-deklarierte, aber nicht definierte Hilfsklasse), der benutzt wird, um die privaten Bestandteile einer Klasse zu verstecken. Das heißt, statt folgendem Code:

```
// file x.h
class X
{
  // public und protected Elemente
private:
  // private Elemente; wann immer sich diese ändern,
  // muss aller Client-Code neu kompiliert werden
};
```

schreiben wir:

1. Pimpl steht für »pointer to implementation«, heißt im Englischen aber auch Pickel. (A. d. Ü.)
2. Ich habe früher immer `impl_` geschrieben. Die Bezeichnung `pimpl_` wurde vor mehreren Jahren durch Jeff Sunner (Chefprogrammierer bei PeerDirect) geprägt, zu gleichen Teilen wegen seines Hangs zur ungarischen Notation mit einem »p« als Präfix für Pointer-Variablen als auch einer gelegentlichen Vorliebe für hässliche Wortspiele.

```
// file x.h
class X
{
  // public und protected Elemente
private:
  struct XImpl* pimpl_;
    // ein Zeiger auf eine vorwärts-deklarierte Klasse
};

// file x.cpp
struct X::XImpl
{
  // private Elemente; vollständig versteckt, können ohne
  // Neukompilierung alles Client-Codes geändert werden
};
```

Jedes `X`-Objekt alloziert dynamisch ein `XImpl`-Objekt. Wenn Sie sich ein Objekt als einen physischen Block vorstellen, so haben wir im Prinzip nichts weiter gemacht, als ein großes Stück dieses Blocks abgetrennt und an dessen Stelle nur eine »kleine Ausbuchtung an einer Seite gelassen – den undurchsichtigen Pointer oder Pimpl.

Die wichtigsten Vorteile dieses Idioms resultieren aus der Tatsache, dass hierdurch Kompilierungsabhängigkeiten durchbrochen werden.

- Typen, die nur in der Klassenimplementierung verwendet werden, müssen nicht länger im Client-Code definiert werden, wodurch zusätzliche `#include`-Direktiven eliminiert werden und die Kompilierung beschleunigt wird.
- Die Implementierung einer Klasse kann geändert werden – das heißt, private Datenelemente können beliebig hinzugefügt oder entfernt werden, ohne dass der Client-Code neu kompiliert werden muss.

Die wichtigsten Nachteile des Idioms sind Performanceeinbußen.

- Bei jedem Erzeugen/Zerstören eines Objekts muss Speicher dynamisch alloziert/freigegeben werden.
- Jeder Zugriff auf eine versteckte Eigenschaft kann einen zusätzlichen Level Indirektion bedeuten. (Wenn eine versteckte Elementfunktion selbst einen Rückwärtszeiger benutzt, um eine Funktion in der sichtbaren Klasse aufzurufen, gibt es noch mehr Indirektionen.)

Wir werden auf diese und andere Problematiken des Pimpl-Idioms später in diesem Abschnitt zurückkommen. In unserem jetzigen Beispiel gibt es drei Headerdateien, die nur für die privaten Datenelemente der Klasse `X` eingebunden werden mussten. Wenn wir `X` mit Hilfe des Pimpl-Idioms umstrukturieren, können wir sofort weitere Vereinfachungen vornehmen.

Benutzen Sie das Pimpl-Idiom, um die privaten Implementierungsdetails von `X` zu verstecken.

```
#include <list>

#include "c.h"  // class C
#imclude "d.h"  // class D
```

Eine der Headerdateien (`c.h`) kann mit einer Vorwärts-Deklaration ersetzt werden, da `C` immer noch als Parameter und Rückgabewert in der öffentlichen Schnittstelle auftaucht. Die anderen beiden Headerdateien (`list` und `d.h`) können komplett verschwinden.

☑ Richtlinie

Benutzen Sie für weithin gebrauchte Klassen das Pimpl-Idiom, um Implementierungsdetails zu verstecken. Verwenden Sie einen undurchsichtigen Zeiger (ein Zeiger auf eine deklarierte, aber nicht definierte Klasse), den Sie als »`struct XxxxImpl* pimpl_;`« deklarieren, um private Eigenschaften (sowohl Datenelemente als auch Elementfunktionen) aufzunehmen. Zum Beispiel:

```
class Map { private: struct MapImpl* pimpl_; };
```

Nach dieser zusätzlichen Änderung sieht die Headerdatei wie folgt aus:

```
//  x.h: nach Anwendung des Pimpl-Idioms
//
#include <iosfwd>
#include "a.h"  // class A (hat virtuelle Funktionen)
#include "b.h"  // class B (hat keine virtuellen Funktionen)
class C;
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  struct XImpl* pimpl_;
    // undurchsichtiger Zeiger auf
    // eine vorwärts-deklarierte Klasse
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Die privaten Details wandern in die Implementierungsdatei von `X`, wo der Client-Code sie niemals sieht und somit auch nie von ihnen abhängt.

```
//  Implementierungsdatei x.cpp
//
struct X::XImpl
{
  std::list<C> clist_;
  D            d_;
};
```

Somit müssen wir nur noch drei Headerdateien einbinden, was eine große Verbesserung darstellt. Aber wie sich herausstellen wird, gibt es noch etwas mehr, was wir tun könnten, wenn es uns erlaubt wäre, die Struktur von `X` noch extremer zu ändern. Was uns sofort zu Lektion 28 führt ...

Lektion 28: Minimieren der Übersetzungszeit-Abhängigkeiten – Teil 3

Schwierigkeitsgrad: 7

Kann, nachdem überflüssige Headerdateien entfernt und unnötige Abhängigkeiten der Interna der Klasse eliminiert wurden, noch mehr entkoppelt werden? Die Anwort führt uns zurück zu einem grundlegenden Prinzip soliden Klassen-Designs.

Wir haben die Headerdatei schon sehr schön auf Vordermann gebracht, aber es könnten immer noch Wege existieren Abhängigkeiten weiter zu reduzieren. Welche weiteren `#include`-Direktiven könnten entfernt werden, wenn wir `X` noch mehr ändern, und wie?

Dieses Mal dürfen Sie alles an `X` ändern außer die öffentliche Schnittstelle, so dass existierender Code, der `X` benutzt, nicht von den Änderungen betroffen ist. Beachten Sie wiederum den Inhalt der Kommentare.

```
//  x.h: nach Anwendung des Pimpl-Idioms,
//       um Implementierungsdetails zu verstecken
//
#include <iosfwd>
#include "a.h"  // class A (hat virtuelle Funktionen)
#include "b.h"  // class B (hat keine virtuellen Funktionen)
class C;
class E;
class X : public A, private B
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
```

```
  virtual std::ostream& print( std::ostream& ) const;
private:
  struct XImpl* pimpl_;
    // undurchsichtiger Zeiger auf
    // eine vorwärts-deklarierte Klasse
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Lösung

Nach meiner Erfahrung richten sich viele C++-Programmierer immer noch nach der Devise: »Es ist nicht OO, wenn Du nicht erbst.« Damit meine ich, dass sie mehr Vererbung einsetzen als eigentlich nötig. Schauen Sie sich noch mal Lektion 24 für einen erschöpfenden Vortrag dazu an, aber die Essenz besteht darin, dass Vererbung (einschließlich – aber nicht nur – der IST-EIN-Beziehung) einfach eine viel stärkere Beziehung ist als eine HAT-EIN- oder BENUTZT-EIN-Beziehung. Wenn Sie Abhängigkeiten managen müssen, sollten Sie immer die Komposition der Vererbung vorziehen oder mit einem umgeschriebenen Zitat von Albert Einstein: »Benutzen Sie eine so starke Beziehung wie nötig, aber keine stärkere.«

In unserem Beispiel erbt `X` öffentlich von `A` und privat von `B`. Rufen Sie sich in Erinnerung, dass öffentliche Vererbung immer eine IST-EIN-Beziehung modellieren und das Liskov-Substitutionsprinzip erfüllen sollte.[3] `X` IST EIN `A` in diesem Fall und da ist auch nichts dran auszusetzen, also belassen wir es auch dabei.

Haben Sie aber das Interessante bei `B` bemerkt?

Das Interessante an `B` ist: `B` ist eine private Basisklasse von `X`, hat aber keine virtuellen Funktionen. Nun ist aber gewöhnlich der einzige Grund, warum man private Vererbung der Komposition vorzieht, der, Zugriff auf `protected`-Elemente der jeweiligen Klasse zu erlangen – was in den meisten Fällen bedeutet, virtuelle Funktionen zu überschreiben.[4] Wie wir sehen, hat `B` keine virtuellen Funktionen, so dass es wahrscheinlich keinen Grund gibt, die stärke Beziehung der Vererbung zu bevorzugen (natürlich nur, falls `X` nicht auf irgendwelche `protected`-Funktionen oder -Daten von `B` zugreifen muss, aber für den Moment wollen wir annehmen, dass dies nicht der Fall ist). Angenommen, dies ist in der Tat der Fall, sollte `X`, anstelle `B` als zusätzliche Basisklasse zu haben,

3. Für eine gute Diskussion zur Anwendung des LSP siehe die Publikationen, die unter `www.objectmentor.com` online verfügbar sind, als auch Martin95.
4. Ja, es gibt andere Gründe für den Einsatz von Vererbung, Situationen dafür sind aber sehr selten und/oder obskur. Eine ausführliche Diskussion der (wenigen) Gründe für den Einsatz beliebiger Vererbung finden Sie in Sutter98(a) und Sutter99. Diese Artikel zeigen detailliert auf, warum anstelle von Vererbung häufig Enthaltung/Komposition angewendet werden sollte.

ein einfaches Objekt vom Typ `B` als Elementvariable einführen. Somit kann die Headerdatei noch weiter vereinfacht werden, indem die unnötige Vererbung von Klasse `B` beseitigt wird.

```
#include "b.h"  // Klasse B (keine virtuellen Funktionen)
```

Da die `B`-Elementvariable `private` sein sollte (schließlich handelt es sich hierbei um ein Implementierungsdetail), sollte diese Elementvariable in dem `pimpl_`-Teil von `X` versteckt werden.

☑ Richtlinie

Benutze niemals Vererbung, wenn Komposition ausreicht.

Dies alles führt uns zu einer gewaltig vereinfachten Headerdatei:

```
//  x.h: nach Beseitigung unnötiger Vererbung
//
#include <iosfwd>
#include "a.h"  // class A
class B;
class C;
class E;
class X : public A
{
public:
     X( const C& );
  B  f( int, char* );
  C  f( int, C );
  C& g( B );
  E  h( E );
  virtual std::ostream& print( std::ostream& ) const;
private:
  struct XImpl* pimpl_; // enthält jetzt heimlich ein B
};
inline std::ostream& operator<<( std::ostream& os, const X& x )
{
  return x.print(os);
}
```

Nach drei Phasen fortschreitender Vereinfachung ist das endgültige Ergebnis, dass `x.h` immer noch überall andere Klassennamen benutzt, aber Benutzer von `X` nur noch für zwei `#include`-Direktiven (`a.h` und `iosfwd`) bezahlen müssen. Was für eine Verbesserung gegenüber der ursprünglichen Version!

Lektion 29: Compiler-Firewalls — Schwierigkeitsgrad: 6

Die Benutzung des Pimpl-Idioms kann Codeabhängigkeiten und Übersetzungszeiten dramatisch reduzieren. Aber was sollte in das `pimpl_`-Objekt wandern und welches ist der sicherste Weg, es zu verwenden?

In C++ muss, wenn sich irgendetwas an der Klassendefinition ändert (selbst `private`-Elemente), alles, was die jeweilige Klasse benutzt, neu kompiliert werden. Zur Reduzierung dieser Abhängigkeiten ist es eine übliche Technik, einen undurchsichtigen Zeiger zu verwenden, um einige der Implementierungsdetails zu verbergen.

```
class X
{
public:
  /* ... public Elemente ... */
protected:
  /* ... protected Elemente? ... */
private:
  /* ... private Elemente? ... */
  struct XImpl* pimpl_;
    // undurchsichtiger Zeiger auf
    // eine vorwärts-deklarierte Klasse
};
```

Die Fragen, die Sie beantworten sollten, sind folgende:

1. Was sollte in `XImpl` wandern? Hierfür gibt es vier gebräuchliche Ansätze:
   - Alle privaten Elementvariablen (aber keine Funktionen) in `XImpl` packen
   - Alle privaten Elemente in `XImpl` packen
   - Alle `private`-und alle `protected`-Elemente in `XImpl` packen oder
   - `XImpl` wird voll und ganz die Klasse, die `X` gewesen wäre, und `X` wird nichts weiter als die öffentliche Schnittstelle, die nur aus Vorwärtsfunktionen besteht (eine Handle-Body-Variante).
2. Benötigt `XImpl` einen Zeiger zurück auf das zugehörige `X`-Objekt?

Lösung

Klasse `X` benutzt eine Variante des Handle-Body-Idioms. Coplien (Coplien92) beschrieb das Handle-Body-Idiom als besonders nützlich zur Referenzzählung gemeinsam benutzter Implementierungen, aber es hat auch eine allgemeinere Verwendung zum Verstecken der Implementierung. Aus Bequemlichkeit werde ich die Klasse `X` im Folgenden die »sichtbare Klasse« und »XImpl« die »Pimpl-Klasse« nennen.

Ein großer Vorteil dieses Idioms ist, dass es Kompilierungsabhängigkeiten aufbricht. Zum einen laufen System-Builds schneller, da durch den Einsatz von Pimpl zusätzliche `#include`-Direktiven eliminiert werden können, wie in Lektionen 27 und 28 gezeigt. Ich habe an Projekten gearbeitet, bei denen nur die Änderung einiger weniger weithin benutzter Klassen, so dass diese Pimpl benutzen, die Übersetzungszeit des Systems halbierte. Zum anderen werden die Auswirkungen von Codeänderungen auf die gesamte Übersetzung reduziert, da alles der Klasse, was sich in dem Pimpl-Teil befindet, frei geändert werden kann – also Elemente beliebig hinzugefügt oder gelöscht werden können –, ohne dass Clientcode neu kompiliert werden muss.

Lassen Sie uns die Fragen dieser Lektion eine nach der andern beantworten.

1. Was sollte in `XImpl` wandern?

   Alternative 1 (6/10 Punkte): Alle privaten Elementvariablen (aber keine Funktionen) in `XImpl` *packen.* Dies ist ein guter Anfang, da wir nun jede Klasse vorwärts deklarieren können, die nur als Elementvariable vorkommt (anstatt die eigentliche Klassendefinition via `#include` einzubinden, wodurch der Clientcode auch von dieser Klasse abhängig würde). Dennoch kann man es meist noch besser machen.

   Alternative 2 (10/10 Punkte): Alle privaten Elemente in `XImpl` *packen.* Dies ist (fast) meine normale Vorgehensweise dieser Tage. Schließlich wird »Clientcode soll sich nicht um diese Sachen kümmern müssen« in C++ durch `private` ausgedrückt – und privat bedeutet der Öffentlichkeit nicht zugänglich.[5]

   Dabei gibt es jedoch einige Punkte zu berücksichtigen, von denen der erste der Grund für das obige »fast« ist.

   – Sie können keine virtuellen Elementfunktionen in der Pimpl-Klasse verstecken, selbst wenn die virtuellen Funktionen private sind. Wenn die virtuelle Funktion eine von einer Basisklasse geerbte Funktion überschreibt, muss sie auch in der eigentlichen abgeleiteten Klasse auftauchen. Wenn die virtuelle Funktion nicht geerbt wurde, muss sie immer noch in der sichtbaren Klasse auftauchen, damit sie von abgeleiteten Klassen überschrieben werden kann.

     Es ist jedoch meistens keine gute Idee, eine virtuelle Funktion privat zu machen. Die Idee hinter einer virtuellen Funktion ist, abgeleiteten Klassen zu erlauben, die Funktion neu zu definieren, und eine verbreitete Technik bei der Neudefinition ist, die geerbte Funktion der Basisklasse aufzurufen (was nicht möglich ist, wenn diese `private` ist), da dort ein Großteil der Funktionalität bereits erledigt wird.

   – Funktionen in der Pimpl-Klasse könnten einen »Rückwärtszeiger« zum sichtbaren Objekt benötigen, wenn sie sichtbare Funktionen des eigentlichen Objektes

5. Außer in einigen liberalen europäischen Staaten.

aufrufen müssen, wodurch außerdem ein zusätzlicher Level der Indirektion eingeführt wird. (Hier bei PeerDirect nennen wir einen solchen Rückwärtszeiger meist `self_`.)

– Oftmals ist der beste Kompromiss, die Alternative 2 zu benutzen und zusätzlich nur solche nicht-privaten Funktionen in `XImpl` zu packen, die von den privaten aufgerufen werden müssen. (Siehe auch die Kommentare zu Rückwärtszeigern weiter unten.)

☑ Richtlinie

Setzen Sie für weithin benutzte Klassen das Compiler-Firewall-Idiom (Pimpl-Idiom) ein, um die Implementierungsdetails zu verstecken. Benutzen Sie einen undurchsichtigen Zeiger (einen Zeiger, der auf eine deklarierte, aber nicht definierte Klasse zeigt), der als »`struct XxxxImpl* pimpl_`;« deklariert wird, um private Elemente (sowohl Variablen als auch Funktionen) zu speichern.

Alternative 3 (0/10 Punkte): Alle `private`-und alle `protected`-Elemente in `XImpl` packen. Zusätzlich alle `protected`-Elemente in die Pimpl-Klasse aufzunehmen ist eigentlich falsch. `protected`-Elemente sollten niemals in die Pimpl-Klasse wandern, da sie dort nur entmannt werden. Schließlich existieren `protected`-Elemente aus dem speziellen Grund, dass sie von abgeleiteten Klassen gesehen und benutzt werden können, womit sie nicht annähernd so nützlich sind, wenn die abgeleitete Klasse sie nicht sehen oder benutzen kann.

Alternative 4 (10/10 Punkte): `XImpl` wird voll und ganz die Klasse, die `X` gewesen wäre, und `X` wird nichts weiter als die öffentliche Schnittstelle, die nur aus Vorwärtsfunktionen besteht (eine andere Handle-Body-Variante). Dies ist in einigen speziellen Fällen sinnvoll und hat den Vorteil, dass kein Rückwärtszeiger benötigt wird, da alle Dienste innerhalb der Pimpl-Klasse zur Verfügung stehen. Der größte Nachteil ist, dass die sichtbare Klasse dadurch nutzlos für Vererbung wird, sowohl als Basisklasse als auch als abgeleitete Klasse.

2. Benötigt `XImpl` einen Zeiger zurück auf das zugehörige `X`-Objekt?

   Benötigt die Pimpl-Klasse einen Rückwärtszeiger auf das sichtbare Objekt? Die Antwort auf diese Frage ist: Manchmal, unglücklicherweise ja. Schließlich machen wir (etwas künstlich) nichts anderes, als jedes Objekt in zwei Hälften zu zerlegen, um eine davon zu verstecken.

   Betrachten Sie Folgendes: Wann immer eine Funktion in der sichtbaren Klasse aufgerufen wird, wird meistens eine Funktion oder Daten in der versteckten Hälfte benötigt, um die Anforderung zu befriedigen. Das ist auch gut und vernünftig. Was aber zunächst vielleicht nicht ganz so offensichtlich ist, ist, dass eine Funktion im Pimpl-Objekt oft eine Funktion der sichtbaren Klasse aufrufen muss, gewöhnlich

weil die aufgerufene Funktion öffentlich oder virtuell ist. Eine Möglichkeit, dies zu umgehen, besteht darin, die Alternative 4 von oben in vernünftiger Weise für die betroffenen Funktionen zu benutzen – das heißt, Alternative 2 zu implementieren und zusätzlich alle nicht-privaten Funktionen in Pimpl zu packen, die von den privaten benutzt werden.

Lektion 30: Das »Fast-Pimpl«-Idiom — Schwierigkeitsgrad: 3

Es ist manchmal verlockend, Kanten abzurunden, unter dem Deckmantel »Abhängigkeiten zu reduzieren« oder »Effizienz«, dies ist aber nicht immer eine gute Idee. Hier finden Sie ein hervorragendes Idiom, um beides gleichzeitig und sicher zu erreichen.

Aufrufe der Standardfunktionen `malloc` und `new` sind relativ kostspielig.[6] Im folgenden Code hat der Programmierer zunächst ein Datenelement vom Typ `X` in die Klasse `Y` gepackt.

```
// Versuch #1
//
// Datei y.h
#include "x.h"
class Y
{
  /*...*/
  X x_;
};

// Datei y.cpp
Y::Y() {}
```

Diese Deklaration der Klasse `Y` erfordert, dass die Deklaration der Klasse `X` (aus `x.h`) beim Kompilieren sichtbar ist. Um dies zu vermeiden, versucht der Programmierer als Erstes Folgendes:

```
// Versuch #2
//
// Datei y.h
class X;
class Y
{
  /*...*/
  X* px_;
};

// Datei y.cpp
```

6. Verglichen mit anderen typischen Operationen wie Funktionsaufrufen.

```
#include "x.h"
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Hiermit ist `X` hübsch versteckt, es stellt sich aber heraus, dass Y eine sehr oft benutzte Klasse ist und der Overhead der dynamischen Speicherallozierung sich negativ auf die Performance auswirkt.

Schließlich kommt unser furchtloser Programmierer auf die »perfekte« Lösung, die weder das Einbinden von `x.h` in `y.h` erfordert noch die Ineffizienz dynamischer Speicherallozierung (und noch nicht mal eine Vorwärts-Deklaration).

```
// Versuch #3
//
// Datei y.h
class Y
{
  /*...*/
  static const size_t sizeofx = /*ein bestimmter Wert*/;
  char x_[sizeofx];
};

// Datei y.cpp
#include "x.h"
Y::Y()
{
  assert( sizeofx >= sizeof(X) );
  new (&x_[0]) X;
}
Y::~Y()
{
  (reinterpret_cast<X*>(&x_[0]))->~X();
}
```

1. Was ist der Speicherplatz-Overhead des Pimpl-Idioms?
2. Was ist der Performance-Overhead des Pimpl-Idioms?
3. Diskutieren Sie Versuch #3. Können Sie sich einen besseren Weg vorstellen, um den Overhead zu vermeiden?

   Bermerkung: Siehe Lektion 29 für mehr Informationen zum Pimpl-Idiom.

Lösung

Lassen Sie uns die Fragen dieser Lektion eine nach der anderen beantworten:

1. Was ist der Speicherplatz-Overhead dieses Pimpl-Idioms?

   »Welcher Speicherplatz-Overhead?«, fragen Sie? Nun, wir brauchen Speicherplatz für wenigstens einen zusätzlichen Zeiger (möglicherweise zwei, wenn es einen

Rückwärtszeiger in `XImpl` gibt) für jedes `X`-Objekt. Dies bedeutet typischerweise wenigstens 4 (oder 8) Bytes auf vielen populären Systemen, manchmal sogar 14 Bytes oder noch mehr in Abhängigkeit von Alingment-Erfordernissen. Probieren Sie das folgende Programm mit Ihrem Lieblingscompiler.

```
struct X { char c; struct XImpl* pimpl_; };
struct X::XImpl { char c; };
int main()
{
  cout << sizeof(X::XImpl) << endl
       << sizeof(X) << endl;
}
```

Bei vielen populären Compilern, die 32-Bit-Zeiger benutzen, führt dies zu folgender Ausgabe:

```
1
8
```

Bei diesen Compilern ist der Overhead zum Speichern des zusätzlichen Zeigers 7 Bytes und nicht 4. Warum? Da die Plattform, auf der der Compiler läuft, es erfordert, dass Zeiger an einer 4-Byte-Grenze ausgerichtet sind, weil ansonsten die Performance, wenn der Zeiger nicht an einer solchen Grenze ausgerichtet wird, bedeutend schlechter wäre. Mit diesem Wissen alloziert der Compiler 3 Bytes unbenutzten/leeren Speichers innerhalb jedes `X`-Objekts, was bedeutet, dass die Kosten für das Hinzufügen eines Zeigers 7 und nicht 4 Bytes beträgt. Wenn außerdem ein Rückwärtszeiger benötigt wird, kann der gesamte Speicherplatz-Overhead bis zu 14 Bytes auf einem 32-Bit-Rechner betragen und 30 Bytes auf einem 64-Bit-Rechner und so weiter.

Wie können wir diesen Speicherplatz-Overhead vermeiden? Die kurze Antwort ist: Wir können ihn nicht eliminieren, können ihn aber manchmal minimieren.

Die lange Antwort ist: Es gibt einen geraden, aber tollkühnen Weg, ihn zu eliminieren, den Sie aber niemals beschreiten sollten (und sagen Sie auch niemandem, dass Sie ihn von mir gehört haben), und es gibt normalerweise einen nicht-portablen, aber korrekten Weg ihn zu minimieren. Die absolut tollkühne »Speicherplatzoptimierung« ist gleichzeitig die absolut tollkühne »Performanceoptimierung«, weshalb ich ihre Diskussion an das Ende dieser Lektion verschoben habe, siehe den Kasten »Tollkühne Optimierungen und warum sie teuflisch sind.«

Wenn (und nur wenn) der Speicherplatzunterschied in Ihrem Programm wirklich von Bedeutung ist, dann ist der nicht-portable, aber korrekte Weg, den Speicherplatz-Overhead zu minimieren, Compiler-spezifische `#pragma`-Direktiven zu benutzen. Viele Kompiler lassen es zu, dass Sie das standardmäßige Alignment/Packing-Verhalten für eine bestimmte Klasse überschreiben; siehe die Compiler-Dokumentation für nähere Informationen hierzu. Wenn Ihre Plattform Zeiger-Alignment nur

»bevorzugt« (und nicht »erfordert«) und Ihr Compiler dieses Feature anbietet, können Sie auf einer 32-Bit-Plattform bis zu 6 Byte Speicherplatz-Overhead pro `X`-Objekt eliminieren, auf Kosten von (wahrscheinlich klitzekleinen) Performance-Einbußen, da die Benutzung des Zeigers nicht mehr so effizient ist. Bevor Sie so etwas aber auch nur in Betracht ziehen, sollten Sie sich immer an die uralte Weisheit halten: *Mach es zuerst richtig, dann schnell.* Optimieren Sie niemals – weder für Geschwindigkeit, noch für Speicherplatz –, bis Ihr Profiler oder andere Werkzeuge Ihnen mitteilen, dass Sie sollten.

2. Was ist der Performance-Overhead des Pimpl-Idioms?

   Die Benutzung des Pimpl-Idioms kann aus zwei Gründen Performance-Einbußen mit sich bringen. Zum einen muss bei jeder Konstruktion/Dekonstruktion eines `X`-Objekts zusätzlich Speicher für das zugehörige `XImpl`-Objekt alloziert/dealloziert werden, was typischerweise eine recht kostspielige Operation ist.[7] Der andere Grund ist, jeder Zugriff auf ein Element in der Pimpl-Klasse kann einen zusätzlichen Level der Indirektion bedeuten; wenn das versteckte Element, auf das zugegriffen wird, selber einen Rückwärtszeiger benutzt, um ein Funktion in der sichtbaren Klasse aufzurufen, gibt es sogar noch mehr zusätzliche Level der Indirektion.

   Wie können wir diesen Performance-Overhead umgehen? Die kurze Antwort ist: Benutzen Sie das Fast-Pimpl-Idiom, das im nächsten Abschnitt vorgestellt wird. (Es gibt auch wieder einen geradlinigen, tollkühnen Weg um den Overhead zu eliminieren, den Sie aber nie benutzen sollten, siehe Kasten »Tollkühne Optimierungen und warum sie teuflisch sind« für weitere Informationen.)

3. Diskutieren Sie Versuch #3.

   Die kurze Antwort zu Versuch #3 ist: Machen Sie so etwas nicht. Es läuft daraus hinaus, dass C++ keine undurchsichtigen Typen direkt unterstützt, und dies ist ein zerbrechlicher Versuch (einige Leute, einschließlich meiner Person, würden dazu auch »Hack« sagen) diese Limitierung zu umgehen.

   Was der Programmierer mit großer Sicherheit möchte, ist etwas anderes, nämlich das Fast-Pimpl-Idiom.

   Der zweite Teil der dritten Frage war: Können Sie sich einen besseren Weg vorstellen, um den Overhead zu vermeiden?

   Der größte Performance-Nachteil hier ist, dass der Speicherplatz für das Pimpl-Objekt aus dem Pool freien Speichers alloziert wird. Der im Allgemeinen richtige Weg, um Performance-Probleme bei der Speicherallozierung für eine bestimmte Klasse zu adressieren, ist, einen klassenspezifischen `operator new()` für diese Klasse

7. Verglichen mit anderen typischen Operationen in C++ wie Funktionsaufrufen. Beachten Sie, dass ich hier über die Kosten des Standard-Allokators spreche, den Sie immer dann benutzen, wenn Sie den eingebauten Operator `::operator new()` oder `malloc()` verwenden.

zur Verfügung zu stellen, der einen Allokator mit fester Größe benutzt, da solche Allokatoren sehr viel effizienter als der Standard-Allokator implementiert werden können.

```
// Datei x.h
class X
{
  /*...*/
  struct XImpl* pimpl_;
};

// Datei x.cpp
#include "x.h"
struct X::XImpl
{
  /*...private Elemente hier...*/
  static void* operator new( size_t )   { /*...*/ }
  static void  operator delete( void* ) { /*...*/ }
};
X::X() : pimpl_( new XImpl ) {}
X::~X() { delete pimpl_; pimpl_ = 0; }
```

»Aha!«, sagen Sie. »Wir haben den heiligen Gral gefunden – das Fast-Pimpl-Idiom«, sagen Sie. Nun, ja, aber warten Sie eine Minute und denken Sie darüber nach, wie dieser Ansatz arbeitet und was er kostet.

In Ihrem Lieblings-Fortgeschrittenen-C++-Buch oder allgemeinen Programmierbuch finden Sie die Details, wie ein effizienter Allokator für Speicherblöcke fester Größe geschrieben werden kann, weshalb ich das hier nicht noch einmal abhandle. Ich werde über die Brauchbarkeit sprechen. Eine Technik ist, die [De-]Allozierungsfunktionen in ein generisches Allokator-Template für feste Speicherplatzgrößen zu packen, vielleicht in etwa so:

```
template<size_t S>
class FixedAllocator
{
public:
  void* Allocate( /*angeforderter Speicherplatz hat immmer Größe S */ );
  void  Deallocate( void* );
private:
  /*...implementiert mit statischen Elementen?...*/
};
```

Da die privaten Implementierungsdetails sehr wahrscheinlich statische Elemente benutzen, könnte es jedoch zu Problemen kommen, wenn `Deallocate` von einem Destruktor eines statischen Objekts aufgerufen wird. Sicherer ist wahrscheinlich ein Singleton, das eine separate Liste für jede angeforderte Speicherplatzgröße verwaltet (oder, aus Gründen der Effizienz, eine separate Liste für jeden angeforderten

Speicherplatzgrößenbereich – zum Beispiel eine Liste für Blöcke der Größe 0-8, eine weitere für Blöcke der Größe 9-16 und so weiter.)

```
class FixedAllocator
{
public:
  static FixedAllocator& Instance();
  void* Allocate( size_t );
  void  Deallocate( void* );
private:
  /*...Singleton-Implementierung, typischerweise
       mit leichter zu verwaltenden statischen Elementen
       als die obige Template-Alternative...*/
};
```

Lassen Sie uns eine Hilfsbasisklasse einführen, um die Aufrufe zu kapseln. Dies funktioniert deshalb, weil abgeleitete Klassen die überladenen Operatoren »erben«.

```
struct FastArenaObject
{
  static void* operator new( size_t s )
  {
    return FixedAllocator::Instance()->Allocate(s);
  }
  static void operator delete( void* p )
  {
    FixedAllocator::Instance()->Deallocate(p);
  }
};
```

Nun können Sie so viele Fast-Pimpl-Klassen schreiben, wie Sie möchten:

```
//  Möchten Sie, dass diese Klasse ein Fast-Pimpl wird?
//  Ganz einfach, leiten Sie sie einfach ab...
struct X::XImpl : FastArenaObject
{
  /*...private Elemente hier...*/
};
```

Wenn wir diese Technik auf das ursprüngliche Problem anwenden, erhalten wir eine Variante von Versuch #2:

```
// Datei y.h
class X;
class Y
{
  /*...*/
  X* px_;
};

// Datei y.cpp
```

```
#include "x.h" // X erbt jetzt von FastArenaObject
Y::Y() : px_( new X ) {}
Y::~Y() { delete px_; px_ = 0; }
```

Passen Sie aber auf! Das ist zwar hübsch, verwenden Sie das Fast-Pimpl-Idiom aber nicht wahllos. Sie erhalten zwar mehr Geschwindigkeit beim Allozieren, aber wie immer sollten Sie auch nicht die Kosten vergessen. Das Verwalten von separaten Listen freier Speicherplatzblöcke bestimmter Größe geht gewöhnlich mit Speicherplatzeinbußen einher, da der freie Speicherplatz (mehr als sonst) über mehrere Listen fragmentiert ist.

Ein letzter Hinweis: Wie auch bei jeder anderen Optimierung, benutzen Sie Pimpl im Allgemeinen und Fast-Pimpl nur in besonderen Fällen, nachdem der Profiler und Ihre Erfahrung gezeigt haben, dass der Extra-Performance-Schub wirklich in dieser Situation benötigt wird.

☑ Richtlinie

Vermeiden Sie `inline`-Funktionen und weiter gehendes Tuning, bis Perfomance-Profiling gezeigt hat, dass dies wirklich notwendig ist.

Tollkühne Optimierungen und warum sie teuflisch sind.

Der Lösungs-Text dieser Lektion hat gezeigt, warum die Benutzung des Pimpl-Idioms Speicherplatz- und Performance-Overhead mit sich bringen kann; er zeigt auch den richtigen Weg, wie diese Overheads minimiert oder eliminiert werden können. Es gibt auch einen manchmal empfohlenen, aber falschen Weg, wie hiermit umzugehen ist.

Hier nun der tollkühne, unsichere, könnte-funktionieren-wenn-Sie-Glück-haben, teuflische, dick machende und für viel Cholesterin sorgende Weg, den Speicherplatz- und Performance-Overhead zu eliminieren, und Sie haben das nicht von mir gehört – der einzige Grund, warum ich ihn hier überhaupt erwähne, ist, weil ich Leute gesehen habe, die ihn probiert haben:

```
// teuflische, heimtückische Headerdatei x.h
class X
{
  /* . . . */
  static const size_t sizeofximpl = /*some value*/;
  char pimpl_[sizeofximpl];
};
// bösartige, verdorbene Implementierungsdatei x.cpp
#include "x.h"
X::X()
```

```
{
  assert( sizeofximpl >= sizeof(XImpl) );
  new (&pimpl_[0]) XImpl;
}
X::~X()
{
  (reinterpret_cast<XImpl*>(&pimpl_[0]))->~XImpl();
}
```

TUN SIE DIES NICHT! Ja, Sie beseitigen so den Speicherplatz-Overhead – es wird kein einziger zusätzlicher Zeiger benötigt.[1] Ja, Sie beseitigen so den Performance-Overhead der Speicherallozierung – es gibt kein einziges `malloc` oder `new`. Ja, es könnte sogar mit der aktuellen Version Ihres aktuellen Compilers funktionieren.

Es ist auch völlig unportabel. Schlimmer noch, es wird Ihr System zum kompletten Absturz bringen, auch wenn es anfangs so aussieht, als würde es funktionieren. Hier einige Gründe:

1. *Alignment.* Für jeden Speicherbereich, der dynamisch mittels `new` oder `malloc` alloziert wird, ist garantiert, dass er für Objekte jedes Typs richtig ausgerichtet ist, was für Puffer, die nicht dynamisch alloziert sind, aber *nicht* der Fall ist:

   ```
   char* buf1 = (char*)malloc( sizeof(Y) );
   char* buf2 = new char[ sizeof(Y) ];
   char  buf3[ sizeof(Y) ];
   new (buf1) Y;     // OK, buf1 dynamisch alloziert (#1)
   new (buf2) Y;     // OK, buf2 dynamisch alloziert (#2)
   new (&buf3[0]) Y; // falsch, buf3 könnte falsch ausgerichtet sein
   (reinterpret_cast<Y*>(buf1))->~Y(); // OK
   (reinterpret_cast<Y*>(buf2))->~Y(); // OK
   (reinterpret_cast<Y*>(&buf3[0]))->~Y(); // falsch
   ```

   Nur um mich ganz klar auszudrücken: Ich empfehle nicht, dass Sie #1 oder #2 verwenden. Ich weise lediglich darauf hin, dass diese beiden Vorgehensweisen legal sind, wogegen der obige Versuch eines Pimpl-Objekts ohne dynamische Speicherallozierung es nicht ist, auch wenn es (gefährlicherweise) so erscheinen mag, dass er zunächst korrekt funktioniert, wenn Sie Glück haben.[2]

2. *Zerbrechlichkeit.* Der Autor von `X` muss übermäßig vorsichtig sein mit ansonsten normalen `X`-Funktionen. Zum Beispiel darf X nicht den Default-Zuweisungsoperator benutzen, sondern entweder Zuweisung unterdrücken oder die Funktionalität selbst bereitstellen. (Das Schreiben eines sicheren `X::operator=` ist nicht allzu schwer, was ich aber dem Leser als Übung überlasse. Erinnern Sie sich daran, hier und in `X::~X` auf Exception-Sicherheit zu achten.[3] Wenn Sie damit fertig sind, denke ich, dass Sie mir zustimmen werden, dass dies mehr Ärger ist, als es die Sache wert ist.)

3. *Wartungskosten.* Wenn `sizeof(XImpl)` größer als `sizeofximpl` wird, muss der Programmierer `sizeofximpl` dementsprechend erhöhen. Dies kann einen sehr unattraktiven Wartungsaufwand darstellen. Die Wahl eines größeren Wertes für `sizeofximpl` kann dies mildern, was aber auf Kosten der Effizienz geht (siehe 4.).
4. *Ineffizienz.* Wann immer `sizeofximpl > sizeof(XImpl)` ist, wird Speicherplatz verschwendet. Dies kann minimiert werden, was aber auf Kosten der Wartbarkeit geht (siehe 3.).
5. *Einfach falsche Dickköpfigkeit.* Kurz gesagt, es ist offensichtlich, dass der Programmierer versucht »etwas Ungewöhnliches« zu tun. Offen gesagt, in meiner Erfahrung ist »ungewöhnlich« fast immer ein Synonym für einen »Hack«. Wann immer Sie etwas in dieser Art sehen – sei es die Allozierung eines Objekts innerhalb eines `char`-Arrays, wie es der Programmierer hier versucht, oder die Implementierung des Zuweisungsoperators mittels explizitem Destruktor-Aufruf und einem Placement-`new`, wie in Lektion 41 diskutiert – sollten Sie einfach NEIN sagen.[4]

Schlussendlich, C++ unterstützt keine unsichtbaren Typen direkt und es ist fragwürdig zu versuchen, diese Limitierung zu umgehen.

1. Die Pimpl-Klasse wird zwar komplett versteckt – aber trotzdem muss aller Client-Code neu kompiliert werden, wenn sich `sizeofximpl` ändert.
2. Ok, ich gebe es zu: Es gibt in der Tat einen (nicht sehr portablen, aber ziemlich sicheren) Weg, die Pimpl-Klasse so wie hier direkt in die Hauptklasse zu packen, wodurch der ganze Speicherplatz- und Performance-Overhead vermieden wird. Es läuft darauf hinaus, dass Sie eine »`max_align`«-Struktur erzeugen, die Ihnen ein maximales Alignment garantiert, und das Pimpl-Element dann als `union { max_align dummy; char pimpl_[sizeofximpl]; }`; definieren – hierdurch wird ein ausreichendes Alignment garantiert. Für die Details suchen Sie nach »`max_align`« im Web oder DejaNews. Wie dem auch sei, ich empfehle Ihnen immer noch dringend, nicht diesen unschönen Weg zu beschreiten, da die Benutzung von `max_align` zwar das erste Problem löst, aber nicht die Probleme zwei bis fünf. Sie wurden gewarnt.
3. Siehe Lektionen 8 bis 17
4. Siehe Lektion 23

6 Namenssuche, Namensräume und das Schnittstellenprinzip

Lektion 31: Namenssuche und das Schnittstellenprinzip – Teil 1

Schwierigkeitsgrad: 9½

Wenn Sie eine Funktion aufrufen, welche Funktion rufen Sie auf? Die Antwort auf diese Frage ist durch die Namenssuche (name lookup) bestimmt, aber Sie werden sicher von einigen Details überrascht sein.

Welche Funktionen werden im folgenden Code aufgerufen? Warum? Analysieren Sie die Implikationen.

```
namespace A
{
  struct X;
  struct Y;
  void f( int );
  void g( X );
}
namespace B
{
  void f( int i )
  {
    f( i );   // welche Funktion f()?
  }
  void g( A::X x )
  {
    g( x );   // welche Funktion g()?
  }
  void h( A::Y y )
  {
    h( y );   // welche Funktion h()?
  }
}
```

Lösung

Zwei der drei Fälle sind (eigentlich) offensichtlich, der dritte jedoch erfordert eine gute Kenntnis der Regeln, wie C++ bei der Namenssuche vorgeht – insbesondere das Koenig-Lookup.

Lassen Sie uns einfach anfangen:

```
namespace A
{
  struct X;
  struct Y;
  void f( int );
  void g( X );
}
namespace B
{
  void f( int i )
  {
    f( i );   // welche Funktion f()?
  }
```

Diese Funktion `f()` ruft sich selbst auf, in unendlicher Rekursion. Der Grund dafür ist, dass die einzige sichtbare Funktion `f() B::f()` selbst ist.

Es gibt eine andere Funktion mit der Signatur `f(int)`, nämlich die im Namensraum `A`. Wenn in `B` »`using namespace A;`« oder »`using A::f;`« geschrieben worden wäre, wäre `A::f(int)` auch ein sichtbarer Kandidat bei der Namenssuche nach `f(int)` gewesen und der Aufruf `f(i)` wäre mehrdeutig zwischen `A::f(int)` und `B::f(int)` gewesen. Da `B` jedoch `A::f(int)` nicht in seinen Namensraum importiert hat, kommt nur `B::f(int)` in Frage und der Aufruf kann eindeutig `B::f(int)` zugeordnet werden.

Und nun die Wendung:

```
  void g( A::X x )
  {
    g( x );   // welche Funktion g()?
  }
```

Dieser Aufruf ist mehrdeutig zwischen `A::g(X)` und `B::g(X)`. Der Programmierer muss beim Aufruf den Namensraum explizit angeben, aus dem er das `g()` benutzen möchte.

Sie mögen sich vielleicht wundern, warum dies so sein sollte. Schließlich wurde wie auch bei `f()` nirgendwo in `B` irgendwo ein »`using`« geschrieben, um `A::g(X)` in den Namensraum zu importieren, so dass man denken sollte, dass nur `B::g(X)` sichtbar ist, nicht wahr? Nun, das wäre auch richtig, wenn es da nicht eine zusätzliche Regel gäbe, die C++ bei der Namenssuche benutzt:

Koenig-Lookup[1] (vereinfacht): *Wenn Sie ein Funktionsargument mit einem Klassentyp (hier x mit dem Typ A::X) angeben, dann betrachtet der Compiler zum Finden des passenden Funktionsnamens auch den Namensraum, der den Argumenttyp enthält (hier A).*

Es ist etwas umfangreicher, aber dies ist die eigentliche Essenz. Hier ein Beispiel direkt aus dem Standard:

```
namespace NS
{
  class T { };
  void f(T);
}
NS::T parm;
int main()
{
  f(parm);    // OK, ruft NS::f auf
}
```

Ich werde jetzt hier nicht näher auf die Gründe eingehen, warum das Koenig-Lookup eine gute Sache ist. (Wenn Sie Ihre Vorstellungskraft jetzt etwas strapazieren wollen, nehmen Sie obigen Code und ersetzen »`NS`« mit »`std`«, »`T`« mit »`string`« und »`f`« mit »`operator<<`« und betrachten Sie die Verzweigungen.) In der nächsten Lektion finden Sie noch weiter gehende Informationen zum Koenig-Lookup und dessen Implikationen für Namensraumisolation und für die Analyse von Klassenabhängigkeiten.

Hier soll es genügen zu sagen, dass das Koenig-Lookup in der Tat eine gute Sache ist und dass Sie sich klar darüber sein sollten, wie es funktioniert, dass es manchmal Einfluss auf den Code haben kann, den Sie schreiben.

Nun zurück zu unserem einfachen Beispiel:

```
  void h( A::Y y )
  {
    h( y );   // welche Funktion h()?
  }
}
```

Es gibt keine andere Funktion `h(A::Y)`, so dass sich diese Funktion selbst aufruft in unendlicher Rekursion.

Obwohl die Signatur von `B::h()` einen Typ aus dem Namensraum `A` enthält, ändert das nichts an dem Ergebnis der Namenssuche, da es keine Funktionen in `A` gibt, deren Name und Signatur mit `h(A::Y)` übereinstimmt.

1. Benannt nach Andrew Koenig, der die Definition festgenagelt hat und seit langer Zeit Mitglied sowohl im AT&T-C++-Team als auch im C++-Standardisierungskomitee ist. Siehe auch Koenig97.

So, was hat das alles zu bedeuten? Dies bringt uns zum Teil der Fragen dieser Lektion: *Analysieren Sie die Implikationen.*

Kurz gesagt, die Bedeutung des Codes im Namensraum `B` wird durch eine Funktion beeinflusst, die sich in einem völlig anderen Namensraum, nämlich `A`, befindet, obwohl in `B` nichts weiter gemacht wurde, als einfach nur einen Typ, der in `A` gefunden wurde, aufzuführen, und es gibt weit und breit keine einzige »`using`«-Anweisung.

Was dies bedeutet, ist, dass Namensräume gar nicht so unabhängig sind, wie die Leute ursprünglich dachten. Fangen Sie jetzt nicht gleich an Namensräume zu denunzieren; denn Namensräume sind immer noch ziemlich unabhängig und erfüllen ihren Zweck immer noch sehr gut. Das Anliegen dieser Lektion ist es, auf einen der (seltenen) Fälle hinzuweisen, in denen Namensräume nicht ganz hermetisch versiegelt sind – und wo sie auch *nicht* hermetisch versiegelt sein sollten, wie die folgende Lektion zeigen wird.

Lektion 32: Namenssuche und das Schnittstellenprinzip – Teil 2 Schwierigkeitsgrad: 9

Was ist in einer Klasse? Das heißt, was ist »Teil« der Klasse und ihrer Schnittstelle?

Rufen Sie sich die traditionelle Definition einer Klasse in Erinnerung:

Eine Klasse beschreibt eine Menge von Daten mit dazugehörigen Funktionen, die auf diesen Daten operieren.

Unsere Frage ist: Welche Funktionen sind »Teil« der Klasse oder definieren die Schnittstelle einer Klasse?

Hinweis #1: Offensichtlich sind nicht-statische Elementfunktionen sehr eng mit der Klasse verbunden und öffentliche nicht-statische Elementfunktionen sind ein Teil der Schnittstelle einer Klasse. Wie verhält es sich mit statischen Elementfunktionen? Und wie steht es mit freien Funktionen?

Hinweis #2: Nehmen Sie sich etwas Zeit die Implikationen von Lektion 31 zu betrachten.

Lösung

Wir werden mit der scheinbar einfachen Frage *»Welche Funktionen sind »Teil« der Klasse oder definieren die Schnittstelle einer Klasse?«* anfangen.

Die tiefer gehenden Fragen sind:

- Wie passt die Antwort mit der objekt-orientierten Programmierung im C-Stil zusammen?

- Wie passt sie mit dem Koenig-Lookup zusammen? Und wie mit Myers Beispiel? (Ich werde beides unten beschreiben.)
- Wie beeinflusst die Antwort die Art und Weise, in der wir Klassenabhängigkeiten analysieren und Objektmodelle entwerfen?

Was also ist in einer Klasse? Hier noch mal die Definition einer »Klasse«:

> *Eine Klasse beschreibt eine Menge von Daten mit dazugehörigen Funktionen, die auf diesen Daten operieren.*
>
> Programmierer missinterpretieren diese Definition häufig unbewusst, indem sie stattdessen sagen: »Oh ja, eine Klasse, das ist das, was in der Klassendefinition steht – die Elementvariablen und Elementfunktionen.« Das ist aber nicht das Gleiche, da hierdurch das Wort *Funktionen* auf nichts außer *Elementfunktionen* eingeschränkt wird. Betrachten Sie folgendes Beispiel:

```
/*** Beispiel 1 (a) ***/
class X { /*...*/ };
/*...*/
void f( const X& );
```

> Die Frage ist: *Ist `f` Teil von `X`?* Einige Leute werden automatisch »Nein« sagen, da `f` keine Elementfunktion (oder eine freie Funktion) ist. Andere mögen etwas fundamental Wichtiges erkennen: Wenn der Code von Beispiel 1 (a) zusammen in einer Headerdatei erscheint, ist er nicht signifikant anders als folgender:

```
/*** Beispiel 1 (b) ***/
class X
{
  /*...*/
public:
  void f() const;
};
```

> Denken Sie einen Moment darüber nach. Abgesehen von den Zugriffsrechten,[2] hat sich an `f` sonst nichts weiter geändert, sie bekommt immer noch einen Zeiger/eine Referenz auf ein X-Objekt. Der `this`-Parameter ist in der zweiten Version nur implizit, das ist alles. So, wenn Beispiel 1 (a) zusammen in einer Headerdatei auftaucht, beginnen wir zu sehen, dass, obwohl `f` keine Elementfunktion von `X` ist, die Funktion dennoch eine sehr enge Beziehung zu `X` hat. Im nächsten Abschnitt zeige ich Ihnen genau, um was für eine Beziehung es sich handelt.

Auf der anderen Seite, wenn `X` und `f` nicht zusammen in der gleichen Headerdatei auftauchen, handelt es sich bei `f` nur um eine alte Client-Funktion und nicht um einen Teil von `X` (auch wenn `f` die Funktionalität von `X` erweitern soll). Wir schreiben routinemä-

2. Selbst die könnten unverändert sein, wenn die ursprüngliche Funktion `f` ein `friend` von `X` wäre.

ßig Funktionen mit Parametern, deren Typen aus Bibliotheks-Headerdateien kommen, und natürlich sind unsere Funktionen kein Bestandteil dieser Bibliotheksklassen.

Mit diesem Beispiel im Hinterkopf möchte ich das folgende Schnittstellenprinzip vorschlagen:

Für eine Klasse `X` *sind alle Funktionen, inklusive freier Funktionen, die sowohl*

- `X` *»aufführen« als auch*
- *zusammen mit* `X` *»zur Verfügung gestellt« werden,*

ein logischer Teil von `X`*, da sie einen Teil der Schnittstelle von X bilden.*

Laut Definition ist jede Elementfunktion »Teil« von X:

- Jede Elementfunktion muss X »aufführen« (eine nicht-statische Elementfunktion hat einen impliziten `this`-Parameter vom Typ `X* const` oder `const X* const`; eine statische Elementfunktion befindet sich in dem Sichtbarkeitsbereich von `X`).
- Jede Elementfunktion muss zusammen mit `X` »zur Verfügung gestellt« werden (innerhalb der Klassendefinition von `X`).

Die Anwendung des Schnittstellenprinzips auf Beispiel 1 (a) führt zu dem gleichen Ergebnis wie unsere ursprüngliche Analyse. Offenbar wird `X` von `f` aufgeführt. Wenn `f` auch mit `X` »zur Verfügung gestellt« wird (zum Beispiel wenn sie zusammen in der gleichen Headerdatei und/oder dem gleichen Namensraum stehen[3]), dann ist entsprechend dem Schnittstellenprinzip `f` ein logischer Teil von `X`, da `f` ein Teil der Schnittstelle von `X` darstellt.

Somit ist das Schnittstellenprinzip ein nützliches Werkzeug, um festzustellen, was wirklich »Teil« einer Klasse ist. Finden Sie es unintuitiv, dass eine freie Funktion als Teil einer Klasse betrachtet werden soll? Dann lassen Sie uns unserem Beispiel etwas mehr Gewicht verleihen, indem wir `f` einen gebräuchlicheren Namen geben.

```
/*** Beispiel 1 (c) ***/
class X { /*...*/ };
/*...*/
ostream& operator<<( ostream&, const X& );
```

Hier ist die Begründung des Schnittstellenprinzips absolut klar, weil wir verstehen, wie diese spezielle freie Funktion arbeitet. Wenn `operator<<` zusammen mit `X` »zur Verfügung gestellt« wird (zum Beispiel in der gleichen Headerdatei/im gleichen Namensraum), dann ist `operator<<` ein logischer Teil von X, da er einen Bestandteil der Schnittstelle von `X` darstellt. Das macht Sinn, obwohl die Funktion keine Elementfunktion von `X` ist, da wir wissen, dass es für Klassenautoren üblich ist, einen `operator<<`

3. Wir betrachten die Beziehung zu Namensräumen gleich genauer, wo sich herausstellen wird, dass sich dieses Schnittstellenprinzip in der gleichen Art und Weise verhält wie das Koenig-Lookup.

bereitzustellen. Wenn der `operator<<` jedoch nicht vom Klassenautoren stammt, sondern von Client-Code, dann ist er auch nicht Bestandteil von `X`, da er auch nicht zusammen mit `X` »zur Verfügung gestellt« wird.[4]

Lassen Sie uns vor diesem Hintergrund zur traditionellen Definition einer Klasse zurückkehren:

> *Eine Klasse beschreibt eine Menge von Daten mit dazugehörigen Funktionen, die auf diesen Daten operieren.*

Diese Definition ist absolut richtig, da sie nirgendwo sagt, ob die »Funktionen« Elementfunktionen sein müssen oder nicht.

Ich habe den C++-Fachausdruck »Namensraum« verwendet, um zu beschreiben, was zusammen mit `X` »zur Verfügung stellen« bedeutet. Ist dies nur spezifisch für das Schnittstellenprinzip so, oder handelt es sich um ein allgemeines OO-Prinzip, das auch in anderen Sprachen angewandt werden kann?

```
/*** Beispiel 2 (a) ***/
struct _iobuf { /*...hier werden die Daten gespeichert...*/ };
typedef struct _iobuf FILE;
FILE* fopen ( const char* filename,
              const char* mode );
int   fclose( FILE* stream );
int   fseek ( FILE* stream,
              long  offset,
              int   origin );
long  ftell ( FILE* stream );
     /* etc. */
```

Dies ist die übliche »Handle-Technik« zum Schreiben von OO-Code in Sprachen, die keine Klassen haben. Sie stellen ein Struktur zur Verfügung, welche die Daten des Objekts enthält, sowie Funktionen – notwendigerweise Nicht-Elementfunktionen –, die einen Zeiger auf diese Struktur erwarten oder zurückgeben. Diese freien Funktionen konstruieren (`fopen`), zerstören (`fclose`) und manipulieren (`fseek`, `ftell` und so weiter) die Daten.

Diese Technik hat Nachteile (zum Beispiel verlässt sie sich darauf, dass Client-Programmierer nicht direkt mit den Daten herumfingern), es handelt sich aber dennoch um »echten« OO-Code – schließlich ist eine Klasse nichts weiter als »eine Menge von Daten mit dazugehörigen Funktionen, die auf diesen Daten operieren.« In diesem Fall sind, da es anders nicht geht, alle Funktionen Nicht-Elementfunktionen, sie sind aber trotzdem Bestandteil der Schnittstelle `FILE`.

4. Die Ähnlichkeit zwischen Element- und Nicht-Elementfunktion ist noch stärker für einige andere überladbare Operatoren. Zum Beispiel wenn Sie »`a+b`« schreiben, könnten Sie sowohl nach `a.operator+(b)` als auch nach `operator+(a,b)` fragen, je nachdem von welchem Typ `a` und `b` sind.

Betrachten Sie nun einen »offensichtlichen« Weg, um Beispiel 2 (a) in einer Sprache neu zu schreiben, die Klassen unterstützt.

```
/*** Beispiel 2 (b) ***/
class FILE
{
public:
  FILE( const char* filename,
        const char* mode );
 ~FILE();
  int  fseek( long offset, int origin );
  long ftell();
       /* etc. */
private:
  /*... hier werden die Daten gespeichert...*/
};
```

Die `FILE*`-Parameter sind nun einfach die impliziten `this`-Parameter geworden. Hier ist sofort klar, dass `fseek` Teil von `FILE` ist, wie es auch in Beispiel 2 (a) war, nur dass es dort eine Nicht-Elementfunktion war. Wir könnten fröhlich einige Elementfunktionen definieren und einige freie Funktionen.

```
/*** Beispiel 2 (c) ***/
class FILE
{
public:
  FILE( const char* filename,
        const char* mode );
 ~FILE();
  long ftell();
       /* etc. */
private:
  /*...data goes here...*/
};
int fseek( FILE* stream,
           long  offset,
           int   origin );
```

Es spielt wirklich keine Rolle, ob die Funktionen Elementfunktionen sind oder nicht. Solange sie `FILE` »aufführen« und mit `FILE` »zur Verfügung gestellt« werden, sind sie auch wirklich Teil von `FILE`. In Beispiel 2 (a) waren alle Funktionen Nicht-Elementfunktionen, weil dies in C so sein muss. Selbst in C++ müssen (oder sollten) einige Funktionen der Schnittstelle einer Klasse Nicht-Elementfunktionen sein. Zum Beispiel kann `operator<<` keine Elementfunktion sein, da er einen Stream als linkes Argument erwartet, und `operator+` sollte keine Elementfunktion sein, damit auch für das linke Argument eine Typumwandlung möglich ist.

Ein tieferer Blick auf das Koenig-Lookup

Das Schnittstellenprinzip macht sogar noch mehr Sinn, da es genau das Gleiche tut wie das Koenig-Lookup. Ich zeige Ihnen zwei Beispiele, um das Koenig-Lookup zu illustrieren und zu definieren.

Im nächsten Abschnitt werde ich Myers Bespiel benutzen, um Ihnen zu zeigen, warum das Koenig-Lookup in direkter Beziehung zum Schnittstellenprinzip steht.

Hier nun die Erklärung, warum wir das Koenig-Lookup benötigen. Es handelt sich um das gleiche Beispiel wie in Lektion 31. Es stammt direkt aus dem C++-Standard.

```
/*** Beispiel 3 (a) ***/
namespace NS
{
  class T { };
  void f(T);
}
NS::T parm;
int main()
{
  f(parm);    // OK, ruft NS::f auf
}
```

Ganz hübsch, oder? »Offensichtlich« sollten Programmierer nicht explizit `NS::f(parm)` schreiben müssen, da nur `f(parm)` »offensichtlich« `NS::f(parm)` bedeutet, nicht wahr? Was aber für uns offensichtlich ist, ist nicht immer so offensichtlich für einen Compiler, insbesondere, wenn man bedenkt, dass nirgendwo eine »`using`«-Anweisung zu finden ist, um den Namen `f` sichtbar zu machen. Das Koenig-Lookup lässt den Compiler an dieser Stelle das Richtige machen.

Und so funktioniert's: Erinnern Sie sich, dass »Namenssuche« nur bedeutet, dass, wann immer Sie einen Aufruf wie »`f(parm)`« schreiben, der Compiler herausfinden muss, welche Funktion mit dem Namen `f` Sie aufrufen wollen. (Durch Überladen und verschiedene Sichtbarkeitsbereiche könnte es mehrere Funktionen mit dem Namen `f` geben.) Das Koenig-Lookup besagt, wenn Sie ein Funktionsargument eines Klassentyps spezifizieren (hier `parm` vom Typ `NS::T`), dann muss der Compiler nicht nur in die üblichen Plätze kucken, wie den lokalen Sichtbarkeitsbereich, sondern auch in den Namensraum, der den Argumenttyp enthält (hier `NS`).[5] Und so funktioniert Beispiel 3 (a): Der Parameter, der an `f` übergeben wird, ist ein `T`, `T` ist im Namensraum `NS` definiert, womit der Compiler die Funktion `f` im Namensraum `NS` in Betracht ziehen kann – ohne Wenn und Aber.

Es ist gut, dass wir `f` nicht explizit qualifizieren müssen, da wir manchmal einen Funktionsnamen *nicht* einfach so qualifizieren können.

5. Der Mechanismus ist etwas komplizierter, aber im Prinzip funktioniert er so.

```
/*** Example 3 (b) ***/
#include <iostream>
#include <string>  // diese Headerdatei deklariert die
                   // Funktion std::operator<< für strings
int main()
{
  std::string hello = "Hello, world";
  std::cout << hello;   // OK: ruft std::operator<< auf
}
```

Hier hätte der Compiler ohne Koenig-Lookup keine Möglichkeit den richtigen `operator<<` zu finden, da der `operator<<`, den wir möchten, eine freie Funktion ist, die uns nur als Teil des `string`-Moduls der Standardbibliothek bekannt ist. Es wäre sehr unschön, wenn der Programmierer diesen Funktionsnamen qualifizieren müsste, da der Operator dann in der letzten Zeile nicht auf natürliche Weise benutzt werden könnte. Stattdessen müssten wir entweder »`std::operator<<( std::cout, hello );`« schreiben, was ausgesprochen hässlich ist, oder »`using namespace std;`«, wodurch alle Namen aus dem Namensraum `std` in den aktuellen Namensraum importiert werden würden, womit aber auch alle Vorteile, die uns Namensräume bieten, praktisch wieder aufgehoben sind. Wenn Ihnen diese beiden Alternativen Schauer den Rücken hinunterjagen, dann wissen Sie, warum wir das Koenig-Lookup benötigen.

Zusammenfassend lässt sich sagen, wenn Sie in dem gleichen Namensraum eine Klasse und eine freie Funktion zur Verfügung stellen, die diese Klasse irgendwie aufführt,[6] stellt der Compiler eine starke Beziehung zwischen beiden her.[7] Und dies bringt uns zurück zum Schnittstellenprinzip, wegen Myers Beispiel.

Mehr Koenig-Lookup: Myers Beispiel

Betrachten Sie zunächst ein (etwas) vereinfachtes Beispiel:

```
/*** Example 4 (a) ***/
namespace NS   // dieser Teil stammt
{              // typischerweise aus
  class T { }; // irgendeiner
}              // Headerdatei T.h
void f( NS::T );
int main()
{
  NS::T parm;
  f(parm);     // OK: ruft die globale Funktion f auf
}
```

6. Als Wert-Parameter, Referenz, Zeiger oder wie auch immer.
7. Zugegeben, die Beziehung ist nicht so stark wie die Beziehung zwischen einer Klasse und einer ihrer Elementfunktionen. Siehe den Kasten »Wie stark ist die ‚Teil-von'-Beziehung« später in dieser Lektion.

Der Namensraum `NS` stellt einen Typ `T` zur Verfügung und der Code außerhalb des Namensraums eine globale Funktion `f`, die ein `T` als Argument erwartet. Das ist schön und gut, der Himmel ist blau, die Welt in Frieden und alles wunderbar.

Es vergeht einige Zeit und eines Tages stellt der Autor von `NS` netterweise eine Hilfsfunktion `f` zur Verfügung:

```
/*** Example 4 (b) ***/
namespace NS   // typischerweise aus
{              // einer Headerdatei T.h
  class T { };
  void f( T ); // <-- neue Funktion
}
void f( NS::T );
int main()
{
  NS::T parm;
  f(parm);     // mehrdeutig: NS::f
}              // oder die globale Funktion f?
```

Das Hinzufügen einer Funktion in den Sichtbarkeitsbereich eines Namensraums hat Code außerhalb des Namensraums zerstört, obwohl der Client-Code nicht einmal `using` benutzt hat, um Namen aus dem Namensraum `NS` zu importieren! Aber halt, es wird noch besser – Nathan Myers wies auf das folgende interessante Verhalten mit Namensräumen und dem Koenig-Lookup hin:

```
/*** Myers Beispiel: "Zuvor" ***/
namespace A
{
  class X { };
}
namespace B
{
  void f( A::X );
  void g( A::X parm )
  {
    f(parm);   // OK: ruft B::f auf
  }
}
```

Das ist schön, der Himmel ist blau. Aber eines Tages fügt der Autor von `A` zur Hilfe seinem Namensraum eine weitere Funktion hinzu:

```
/*** Myers Beispiel: "Danach" ***/
namespace A
{
  class X { };
  void f( X ); // <-- neue Funktion
}
namespace B
```

```
{
  void f( A::X );
  void g( A::X parm )
  {
    f(parm);   // mehrdeutig: A::f oder B::f?
  }
}
```

»Was?«, werden Sie sich fragen. »Die ganze Idee von Namensräumen ist, Namenskollisionen zu verhindern, oder etwa nicht? Das Hinzufügen einer Funktion in einem Namensraum scheint offenbar Code in einem völlig anderen Namensraum zu, zerstören'«. Richtig, der Code im Namensraum `B` scheint zerstört zu werden, bloß weil er einen Typ aus dem Namensraum `A` aufführt. Nirgendwo im Code von `B` wurde »`using namespace A;`« geschrieben und auch nicht »`using A::X;`«.

Dies ist aber kein Problem und `B` ist auch nicht zerstört. Dis ist in der Tat *genau* das, was passieren soll.[8] Wenn es eine Funktion `f(X)` im gleichen Namensraum wie `X` gibt, dann – entsprechend dem Schnittstellenprinzip – ist `f` Teil der Schnittstelle von `X`. Es spielt überhaupt keine Rolle, dass `f` eine freie Funktion ist; um klar zu machen, dass es ein logischer Teil von `X` ist, geben wir ihr einfach einen anderen Namen:

```
/*** Myers Beispiel umgeschrieben: "Danach" ***/
namespace A
{
  class X { };
  ostream& operator<<( ostream&, const X& );
}
namespace B
{
  ostream& operator<<( ostream&, const A::X& );
  void g( A::X parm )
  {
    cout << parm; // mehrdeutig: A::operator<< oder
  }               //             B::operator<<?
}
```

Wenn Client-Code eine Funktion zur Verfügung stellt, die `X` aufführt und mit der Signatur einer Funktion übereinstimmt, die im gleichen Namensraum wie `X` zur Verfügung gestellt wird, dann *soll* der Aufruf mehrdeutig sein. In `B` *soll* es notwendig sein, zu sagen, welche der beiden Funktionen wirklich gemeint ist, die im gleichen Namensraum oder die, die im Namensraum von `X` zur Verfügung gestellt wird. Dies ist genau das Verhalten, das wir mit dem gegebenen Schnittstellenprinzip erwarten sollten:

Für eine Klasse `X` *sind alle Funktionen, inklusive freier Funktionen, die sowohl*

- `X` *»aufführen« als auch*

8. Genau dieses Beispiel kam im Morristown-Treffen im November 1997 auf und es war dieses Beispiel, was mich zum Nachdenken über dieses Thema gebracht hat.

- *zusammen mit* `X` *»zur Verfügung gestellt« werden,*

ein logischer Teil von `X`*, da sie einen Teil der Schnittstelle von X bilden.*

Was Myers Beispiel wirklich bedeutet, ist, dass Namensräume nicht ganz so unabhängig sind, wie Leute ursprünglich gedacht haben, aber sie sind immer noch unabhängig genug, um ihrem eigentlichen Zweck dienen zu können.

Kurz gesagt, es ist kein Zufall, dass das Schnittstellenprinzip genauso wie das Koenig-Lookup funktioniert. Das Koenig-Lookup funktioniert so, wie es funktioniert, im Grunde genommen *aufgrund* des Schnittellenprinzips.

Der Kasten »Wie stark ist die ‚Teil-von'-Beziehung« zeigt, warum eine Elementfunktion immer noch in einer stärkeren Beziehung zu einer Klasse verglichen zu einer freien Funktion steht.

Wie stark ist die »Teil-von«-Beziehung

Auch wenn das Schnittstellenprinzip die Aussage macht, dass sowohl Elementfunktionen als auch Nicht-Elementfunktionen logischer »Teil« einer Klasse sein können, behauptet es nicht, dass Elementfunktionen und Nicht-Elementfunktionen gleichwertig sind. Zum Beispiel haben Elementfunktionen automatisch vollständige Zugriffsrechte auf Klasseninterna, während Nicht-Elementfunktionen solche Zugriffsrechte nur erlangen, wenn sie als `friend` der Klasse definiert werden. Ähnlich gilt für die Namenssuche, inklusive dem Koenig-Lookup, dass C++ absichtlich festlegt, dass eine Elementfunktion in einer stärkeren Beziehung zur Klasse stehend betrachtet werden soll als eine Nicht-Elementfunktion.

```
/*** NICHT Myers Beispiel ***/
namespace A
{
  class X { };
  void f( X );
}
class B      // <-- Klasse, kein Namensraum
{
  void f( A::X );
  void g( A::X parm )
  {
    f(parm); // OK: B::f, nicht mehrdeutig
  }
};
```

Nun, da wir von einer Klasse `B` statt eines Namensraums `B` sprechen müssen, gibt es keine Mehrdeutigkeit mehr. Wenn der Compiler eine passende Elementfunktion mit dem Namen `f()` findet, kümmert er sich nicht weiter darum, eine freie Funktion mit Hilfe des Koenig-Lookups zu finden.

Somit ist an zwei wichtigen Punkten – Zugriffsrechte und Namenssuche-Regeln – festgelegt, dass obwohl eine Funktion entsprechend dem Schnittstellenprinzip »Teil« einer Klasse sein kann, eine Elementfunktion in einer stärkeren Beziehung zu ihrer Klasse steht, als eine Nicht-Elementfunktion.

Lektion 33: Namenssuche und das Schnittstellenprinzip – Teil 3 Schwierigkeitsgrad: 5

Nehmen Sie sich einige Minuten Zeit und betrachten Sie einige der Auswirkungen des Schnittstellenprinzips auf die Art und Weise, wie wir über Programmentwürfe nachdenken. Wir greifen ein bekanntes Klassenentwurfsproblem neu auf: »Wie wird der `operator<<()` *am besten geschrieben?«*

Es gibt zwei Hauptwege, wie der operator<<() für eine Klasse geschrieben werden kann: als eine freie Funktion, die nur die gewöhnliche Schnittstelle der Klasse benutzt (und vielleicht als `friend` auch direkt auf nicht-öffentliche Teile der Klasse zugreift), oder als eine freie Funktion, die eine Hilfsfunktion der Klasse in der Art von `virtual print()` aufruft.

Welche Methode ist die bessere? Was sind die Nachteile?

Lösung

Sie wundern sich bestimmt, warum eine Frage wie diese die Überschrift »Namenssuche – Teil 3« bekommen hat? Falls dies der Fall ist, werden Sie bald sehen warum, wenn wir die Anwendung des Schnittstellenprinzips betrachten, die in der vorigen Lektion diskutiert wurde.

Wovon hängt eine Klasse ab?

»Was ist in einer Klasse?«, ist nicht nur eine rein philosophische Frage. Es ist eine absolut praxisrelevante Frage, da wir ohne die richtige Antwort auf diese Frage Klassenabhängigkeiten nicht richtig analysieren können.

Um dies zu demonstrieren betrachten wir ein scheinbar völlig anderes Problem: *»Wie wird der operator<<() am besten geschrieben?«* Es gibt zwei Möglichkeiten, die beide Nachteile mit sich bringen. Wir werden beide analysieren. Am Ende werden wir sehen, dass wir wieder beim Schnittstellenprinzip angekommen sind und dieses uns eine wichtige Führungshilfe bei der Analyse der Vor- und Nachteile war.

Hier die erste Möglichkeit:

```
/*** Beispiel 5 (a) - nicht-virtuelle Ausgabe ***/
class X
{
  /*...ostream wird hier nirgendwo erwähnt...*/
};
ostream& operator<<( ostream& o, const X& x )
{
  /* Code zur Ausgabe eines X auf einen Stream */
  return o;
}
```

Und hier die zweite:

```
//*** Beispiel 5 (b) -- virtuelle Ausgabe ***/
class X
{
  /*...*/
public:
  virtual ostream& print( ostream& );
};
ostream& X::print( ostream& o )
{
  /* Code zur Ausgabe eines X auf einen Stream */
  return o;
}
ostream& operator<<( ostream& o, const X& x )
{
  return x.print( o );
}
```

Nehmen Sie in beiden Fällen an, dass sowohl die Klasse als auch die Funktionsdeklaration in der gleichen Headerdatei und/oder dem gleichen Namensraum erscheinen. Welche Möglichkeit würden Sie wählen? Was sind die Vor- und Nachteile? Alte, erfahrene C++-Programmierer haben diese Möglichkeiten wie folgt analysiert:

- Möglichkeit (a) hat den Vorteil (*haben wir bis jetzt behauptet*), dass `X` weniger Abhängigkeiten hat. Da keine Elementfunktion von `X` irgendwo `ostream` aufführt, hängt `X` nicht von `ostream` ab (*es scheint so*). Möglichkeit (a) vermeidet auch den Overhead eines zusätzlichen virtuellen Funktionsaufrufs.
- Der Vorteil von Möglichkeit (b) ist, dass jede von `X` abgeleitete Klasse auch korrekt ausgegeben werden kann, selbst wenn ein `X&` an den `operator<<` übergeben wird.

Dies ist die traditionelle Analyse. Soll heißen, die Analyse ist mangelhaft. Ausgerüstet mit dem Schnittstellenprinzip sehen wir warum: Der erste Vorteil von Möglichkeit (a) ist ein Phantom, wie durch die kursiven Bemerkungen bereits angedeutet wurde.

1. Entsprechend dem Schnittstellenprinzip, solange ein `operator<<` sowohl `X` »aufführt« (in beiden Fällen wahr) als auch mit `X` »zur Verfügung gestellt« wird (ebenfalls in beiden Fällen wahr), ist er ein logischer Teil von `X`.

2. In beiden Fällen führt `operator<<` `ostream` auf, somit hängt der `operator<<` von ostream ab.

3. Da in beiden Fällen `operator<<` ein logischer Teil von `X` ist und `operator<<` von `ostream` abhängt, hängt somit auch in beiden Fällen `X` von `ostream` ab.

   Somit ist der Hauptvorteil, den wir traditionell bei Möglichkeit (a) gesehen haben, überhaupt kein Vorteil. In beiden Fällen hängt `X` trotzdem von `ostream` ab. Wenn, wie es typischerweise der Fall ist, `operator<<` und `X` in der gleichen Headerdatei `X.h` erscheinen, dann hängen sowohl das Implementierungsmodul von `X` als auch alle Client-Module, die `X` physikalisch benutzen, von `ostream` ab und erfordern zumindest die Vorwärtsdeklaration von `ostream`, damit sie erfolgreich kompiliert werden können.

   Nachdem der Hauptvorteil von Möglichkeit (a) als Phantom entblößt wurde, ist für die Wahl wirklich nur der Overhead des virtuellen Funktionsaufrufs von Bedeutung. Ohne Anwendung des Schnittstellenprinzips wären wir jedoch nicht so leicht in der Lage gewesen, die wahren Abhängigkeiten zu analysieren (und damit die wirklichen Vorteile) in diesem Beispiel aus der realen Programmierwelt.

   Schlussendlich, es ist nicht immer sinnvoll, zwischen Elementfunktionen und Nicht-Elementfunktionen zu unterscheiden, insbesondere wenn es um die Analyse von Abhängigkeiten geht, und genau das ist es, worum es beim Schnittstellenprinzip geht.

Einige interessante (und überraschende) Ergebnisse

Im Allgemeinen gilt, wenn `A` und `B` Klassen sind und `f(A,B)` ein freie Funktion:

- Wenn `A` und `f` zusammen zur Verfügung gestellt werden, dann ist `f` Teil von `A`, womit `A` von `B` abhängt.
- Wenn `B` und `f` zusammen zur Verfügung gestellt werden, dann ist `f` Teil von `B`, womit `B` von `A` abhängt.
- Wenn `A`, `B` und `f` zusammen zur Verfügung gestellt werden, dann ist `f` sowohl Teil von `A` als auch von `B`, womit `A` und `B` gegenseitig voneinander abhängen. Dies hat schon seit langem auf einem instinktiven Level Sinn gemacht – wenn ein Bibliotheksprogrammierer zwei Klassen zur Verfügung stellt und eine Operation, die beide Klassen benutzt, dann sind die drei wahrscheinlich dafür gedacht, zusammen benutzt zu werden. Nun hat uns das Schnittstellenprinzip ein Mittel in die Hand gegeben, diese Abhängigkeit klarer ausdrücken zu können.

Zum Schluss kommen wir zu den wirklich interessanten Fällen. Wenn `A` und `B` Klassen sind und `A::g(B)` eine Elementfunktion von `A` ist, dann gilt im Allgemeinen:

- Da es die Funktion `A::g(B)` gibt, hängt `A` offensichtlich von `B` ab. Soweit keine Überraschung.
- Wenn `A` und `B` zusammen zur Verfügung gestellt werden, werden natürlich auch `A::g(B)` und `B` zusammen zur Verfügung gestellt. Aus der Tatsache, dass `A::g(B)` sowohl `B` »aufführt« als auch zusammen mit `B` »zur Verfügung gestellt« wird, folgt entsprechend dem Schnittstellenprinzip (vielleicht zunächst überraschend), dass `A::g(B)` ein Teil von B ist, und da `A::g(B)` einen (impliziten) `A*`-Parameter hat, hängt `B` von `A` ab. Da `A` auch von `B` abhängt, bedeutet dies, dass `A` und `B` gegenseitig voneinander abhängen.

Zunächst mag es merkwürdig anmaßen, dass eine Elementfunktion einer Klasse auch als Teil einer anderen Klasse zu betrachten ist, dies ist aber nur der Fall, wenn `A` und `B` *zusammen zur Verfügung gestellt* werden. Betrachten Sie Folgendes: Wenn `A` und `B` zusammen zur Verfügung gestellt werden (sagen wir, in der gleichen Headerdatei) und `A` erwähnt `B` in einer Elementfunktion wie in `A::g(B)`, dann sagt uns unser Bauchgefühl bereits, dass `A` und `B` wahrscheinlich gegenseitig voneinander abhängen. Sie sind auf alle Fälle stark miteinander verbunden und interagieren miteinander, was bedeutet: (a) es ist beabsichtigt, dass sie zusammen verwendet werden, und (b), dass Änderungen an einer Klasse auch die andere betreffen.

Das Problem war bis jetzt, dass die gegenseitige Abhängigkeit von `A` und `B` schwer zu beweisen war außer durch unser Bauchgefühl. Nun kann ihre gegenseitige Abhängigkeit als direkte Konsequenz aus dem Schnittstellenprinzip hergeleitet werden.

Beachten Sie, dass im Gegensatz zu Klassen Namensräume nicht in einem Stück vollständig deklariert werden müssen, und damit hängt »zusammen zur Verfügung gestellt« davon ab, welche Teile des Namensraums sichtbar sind.

```
/*** Beispiel 6 (a) ***/
//---file a.h---
namespace N { class B; } // Vorwärtsdeklaration
namespace N { class A; } // Vorwärtsdeklaration
class N::A { public: void g(B); };
//---file b.h---
namespace N { class B { /*...*/ }; }
```

Programmierer, die `A` benutzen wollen, binden `a.h` ein, womit für sie `A` und `B` zusammen zur Verfügung gestellt werden und gegenseitig voneinander abhängen. Clients von `B` binden `b.h` ein, für sie werden `A` und `B` nicht zusammen zur Verfügung gestellt.

Zusammenfassend möchte ich, dass Sie drei Gedanken aus dieser Lektion mitnehmen.

1. Das Schnittstellenprinzip: Für eine Klasse `X` sind alle Funktionen, einschließlich freier Funktionen, die `X` »aufführen« und mit `X` »zur Verfügung gestellt« werden, logischer Teil von `X`, da sie einen Teil der Schnittstelle von `X` darstellen.

2. Deshalb können sowohl Elementfunktionen als *auch Nicht-Elementfunktionen* logischer »Teil« einer Klasse sein. Eine Elementfunktion steht jedoch immer noch in einer stärkeren Beziehung zu ihrer Klasse als eine Nicht-Elementfunktion.

3. Für das Schnittstellenprinzip ist eine gute Möglichkeit »zusammen zur Verfügung gestellt« zu interpretieren als »in der gleichen Headerdatei und/oder im gleichen Namensraum«. Wenn die Funktion in der gleichen Headerdatei wie die Klasse auftaucht, ist sie von den Abhängigkeiten her betrachtet »Teil« der Klasse. Wenn die Funktion im gleichen Namensraum wie die Klasse steht, ist sie »Teil« der Klasse bezogen auf die Benutzung von Objekten und der Namenssuche.

Lektion 34: Namenssuche und das Schnittstellenprinzip – Teil 4

Schwierigkeitsgrad: 9

Wir beenden diese kleine Serie mit der Betrachtung einiger Implikationen des Schnittstellenprinzips auf die Namenssuche. Können Sie das (ziemlich subtile) Problem ausmachen, das im folgenden Code schlummert?

1. Was sind versteckte Namen (*name hiding*)? Zeigen Sie, wie hierdurch die Sichtbarkeit von Namen aus einer Basisklasse in abgeleiteten Klassen beeinflusst werden kann.

2. Wird der folgende Code korrekt kompiliert? Geben Sie eine so ausführliche Antwort wie möglich. Versuchen Sie, alle Problemgebiete zu isolieren und zu erklären.

```
// Beispiel 1: wird dies kompiliert?
//
// In einer Bibliotheks-Headerdatei:
//
namespace N { class C {}; }
int operator+(int i, N::C) { return i+1; }
//
// eine main-Funktion zum Ausprobieren:
//
#include <numeric>
int main()
{
  N::C a[10];
  std::accumulate(a, a+10, 0);
}
```

Lösung

Lassen Sie uns ein bekanntes Thema der Vererbung – versteckte Namen – auffrischen, indem wir die erste Frage dieser Lektion beantworten:

1. Was sind versteckte Namen (*name hiding*)? Zeigen Sie, wie hierdurch die Sichtbarkeit von Namen aus einer Basisklasse in abgeleiteten Klassen beeinflusst werden kann.

Versteckte Namen

Betrachten Sie folgendes Bespiel:

```
// Beispiel 1a: Verstecken eines Namens
//              aus einer Basisklasse
//
struct B
{
  int f( int );
  int f( double );
  int g( int );
};
struct D : public B
{
private:
  int g( std::string, bool );
};
D   d;
int i;
d.f(i);  // ok, bedeutet B::f(int)
d.g(i);  // falsch: g erwartet 2 Argumente
```

Die meisten von uns sollten mit diesem Problem eines versteckten Namens vertraut sein, obwohl die Tatsache, dass die letzte Zeile nicht kompiliert wird, die meisten neuen C++-Programmierer überrascht. Kurz gesagt, wenn wir eine Funktion mit dem Namen `g` in der abgeleiteten Klasse `D` deklarieren, werden dadurch alle Funktionen mit dem gleichen Namen in allen direkten und indirekten Basisklassen versteckt. Es spielt überhaupt keine Rolle, dass `D::g` »offensichtlich« nicht die Funktion sein kann, die der Programmierer aufrufen wollte (`D::g` hat nicht nur eine falsche Signatur, sondern ist auch `private`, weshalb auf sie gar nicht zugegriffen werden kann). `B::g` ist versteckt und kann bei der Namenssuche nicht berücksichtigt werden.

Um zu sehen, was genau vor sich geht, lassen Sie uns etwas mehr ins Detail gehen, was der Compiler tut, wenn er auf den Funktionsaufruf `d.g(i)` stößt. Als Erstes sieht er in dem aktuellen Sichtbarkeitsbereich nach, in diesem Fall den Sichtbarkeitsbereich der Klasse `D`, und erstellt eine Liste aller Funktionen, die er finden kann und die den

Namen `g` haben (egal ob auf sie zugegriffen werden kann oder ob sie die richtige Anzahl an Argumenten haben). *Nur wenn der Compiler keine einzige Funktion findet*, setzt er die Namenssuche in den nächstäußeren Sichtbarkeitsbereich fort – in diesem Fall der Sichtbarkeitsbereich von Klasse `B` – und wiederholt diese Prozedur, bis es entweder keine weiteren Sichtbarkeitsbereiche mehr gibt, ohne dass er eine Funktion mit dem richtigen Namen gefunden hat, oder er findet einen Sichtbarkeitsbereich, der wenigstens einen Funktionskandidaten enthält, woraufhin der Compiler die Suche beendet und mit den Kandidaten, die gefunden worden weiterarbeitet. Erst hier löst er überladene Funktionen auf und überprüft die Zugriffsrechte.

Es gibt viele gute Gründe, warum die Sprache so arbeiten muss.[9] Um den Extremfall zu nehmen, es macht intuitiv Sinn, dass eine Elementfunktion, die fast genau mit der erforderlichen Signatur übereinstimmt, einer globalen Funktion vorgezogen werden sollte, die ganz genau passt, wenn nur die Typen der Parameter betrachtet werden.

Wie umgeht man versteckte Namen?

Natürlich gibt es zwei gebräuchliche Wege um das Problem mit dem versteckten Namen aus Beispiel 1a zu umgehen. Als Erstes kann im aufrufenden Code einfach gesagt werden, welche Funktion gewünscht ist, wodurch der Compiler gezwungen wird, den richtigen Sichtbarkeitsbereich zu betrachten.

```
// Beispiel 1b: Anfordern eines Namens
//              aus einer Basisklasse
//
D   d;
int i;
d.f(i);    // ok, bedeutet B::f(int)
d.B::g(i); // ok, fordert B::g(int) an
```

Die zweite und gewöhnlich bessere Methode ist, dass der Programmierer von `D` die geerbte Funktion `B::g` durch eine `using`-Anweisung sichtbar machen kann. Dies erlaubt dem Compiler `B::g` im gleichen Sichtbarkeitsbereich wie `D::g` zu betrachten, wenn er nach Funktionen mit dem passenden Namen sucht und anschließend die überladenen Funktionen auflöst.

9. Zum Beispiel könnte man denken, wenn keine der Funktionen in einem inneren Sichtbarkeitsbereich brauchbar ist, dass der Compiler die Suche in den nächstäußeren Sichtbarkeitsbereichen fortsetzen könnte. Das würde jedoch in einigen Fällen zu sehr überraschenden Ergebnissen führen. (Betrachten Sie den Fall, wenn es in einem äußeren Sichtbarkeitsbereich eine genau passende Funktion gibt, es aber in einem inneren Sichtbarkeitsbereich eine Funktion gibt, die fast passt und nur ein paar Parameterumwandlungen erfordert.) Man könnte auch denken, dass der Compiler doch einfach eine Liste aller Funktionen mit dem passenden Namen aus allen Sichtbarkeitsbereichen machen könnte und erst danach überladene Funktionen auflösen und Zugriffsrechte überprüfen sollte. Aber auch hier gibt es einige Stolperstricke. (Bedenken Sie, dass eine Elementfunktion einer globalen Funktion vorgezogen werden und nicht zu einem mehrdeutigen Funktionsaufruf führen sollte.)

```
// Beispiel 1c: Sichtbarmachen eines Names
//              aus einer Basisklasse
//
struct D : public B
{
  using B::g;
private:
  int g( std::string, bool );
};
```

Beide Möglichkeiten bieten einen Weg, um das Problem mit dem versteckten Namen aus Beispiel 1a zu lösen.

Namensräume und das Schnittstellenprinzip

Benutzen Sie Namensräume weise. Wenn Sie eine Klasse in einen Namensraum packen, stellen Sie sicher, dass Sie auch alle Hilfsfunktionen und Operatoren in den gleichen Namensraum stellen. Sollten Sie dies nicht tun, könnten Sie einige überraschende Effekte in Ihrem Code entdecken.

Das folgende einfache Programm basiert auf Code, der mir von dem scharfsinnigen Leser Darin Adler per E-Mail zugesandt wurde. Es stellt eine Klasse `C` im Namensraum `N` zur Verfügung sowie eine Operation auf dieser Klasse. Beachten Sie, dass der `operator+()` sich im globalen Namensraum befindet und nicht im Namensraum `N`. Spielt dies eine Rolle? Ist der Code nicht trotzdem gültig, so wie er geschrieben ist?

Frage 2 war, wie Sie sich erinnern werden:

2. Wird der folgende Code korrekt kompiliert? Geben Sie eine so ausführliche Antwort wie möglich. Versuchen Sie alle Problemgebiete zu isolieren und zu erklären.

```
// Beispiel 2: wird dies kompiliert?
//
// In einer Bibliotheks-Headerdatei:
//
namespace N { class C {}; }
int operator+(int i, N::C) { return i+1; }
//
// eine main-Funktion zum Ausprobieren:
//
#include <numeric>
int main()
{
  N::C a[10];
  std::accumulate(a, a+10, 0);
}
```

Bevor Sie weiterlesen, halten Sie einen Moment inne und überlegen Sie sich mit den Hinweisen, die ich Ihnen bis jetzt gegeben habe: Wird dieses Programm kompiliert? Ist es portabel?[10]

Versteckte Namen in verschachtelten Namensräumen

Nun, auf den ersten Blick sieht Beispiel 2 sicherlich legal aus. Deshalb ist die Antwort wahrscheinlich überraschend: Vielleicht wird es kompiliert, vielleicht auch nicht. Dies hängt völlig von Ihrer Implementierung der Standard-Bibliothek ab, und ich kenne standardkonforme Implementierungen, mit denen das Programm korrekt kompiliert wird, und gleichermaßen standardkonforme Implementierungen, mit denen dies nicht der Fall ist. Kommen Sie ein bisschen näher und ich werde Ihnen sagen warum.

Der Schlüssel, um die Antwort verstehen zu können, ist, zu verstehen, was der Compiler innerhalb von `std::accumulate` machen muss. Die Standard-Templatefunktion `std::accumulate` könnte in etwas so aussehen:

```
namespace std
{
  template<class Iter, class T>
  inline T accumulate( Iter first,
                       Iter last,
                      T    value )
  {
    while( first != last )
    {
      value = value + *first;  // 1
      ++first;
    }
    return value;
  }
}
```

Der Code in Beispiel 2 ruft genau genommen `std::accumulate<N::C*,int>` auf. Wie soll der Compiler in der oben mit 1 kommentierten Zeile den Ausdruck `value + *first` interpretieren? Nun, er muss nach einem `operator+()` suchen, der `int` und `N::C` als Parameter hat (oder Parameter, die in `int` und `N::C` konvertiert werden können). Hey, genau so einen Operator, nämlich `operator+(int,N::C)`, haben wir ja im globalen Namensraum! Ja, da ist er! Cool. Somit muss doch alles in Ordnung sein, oder?

10. Falls Sie sich fragen, ob es ein potenzielles Portabilitätsproblem gibt in Abhängigkeit, ob die Implementierung von `std::accumulate()` den `operator+(int,N::C)` oder den `operator+(N::C,int)` aufruft: Es gibt kein solches Problem. Der Standard besagt, dass Ersteres der Fall ein muss, somit stellt Beispiel 2 einen `operator+()` mit der richtigen Signatur zur Verfügung.

Das Problem ist, dass der Compiler vielleicht nicht in der Lage ist, den `operator+(int,N::C)` im globalen Namensraum zu sehen, in Abhängigkeit davon, welche anderen Funktionen zu dem Zeitpunkt bereits im Namensraum `std` gesehen wurden, wenn `std::accumulate<N::C*,int>` instanziiert wird.

Um zu sehen warum, bedenken Sie, dass der gleiche Mechanismus der versteckten Namen, den wir im Beispiel 1a mit der abgeleiteten Klasse gesehen haben, für alle verschachtelten Blöcke zutrifft, einschließlich Namensräume. Und nun überlegen Sie sich, wo der Compiler anfängt, nach dem `operator+()` zu suchen. (Ich benutze jetzt wieder meine Erklärung aus dem vorigen Abschnitt, nur dass ein paar Namen vertauscht sind). Als Erstes sieht er im aktuellen Sichtbarkeitsbereich nach, in diesem Fall im Sichtbarkeitsbereich des Namensraums `std`, und erstellt eine Liste aller Funktionen, die er finden kann und die den Namen `oerpator+` haben (egal ob auf sie zugegriffen werden kann oder ob sie die richtige Anzahl an Argumenten haben). *Nur wenn der Compiler keine einzige Funktion findet*, setzt er die Namenssuche im nächstäußeren Sichtbarkeitsbereich fort – in diesem Fall im Sichtbarkeitsbereich des den Namensraum `std` umgebenden Namensraums, welches der globale Namensraum ist – und wiederholt diese Prozedur, bis es entweder keine weiteren Sichtbarkeitsbereiche mehr gibt, ohne dass er eine Funktion mit dem richtigen Namen gefunden hat, oder er findet einen Sichtbarkeitsbereich, der wenigstens einen Funktionskandidaten enthält, woraufhin der Compiler die Suche beendet und mit den Kandidaten, die gefunden wurden, weiterarbeitet. Erst hier löst er überladene Funktionen auf und überprüft die Zugriffsrechte.

Kurz gesagt, ob Beispiel 2 kompiliert wird oder nicht, hängt vollständig davon ab, ob Ihre Implementierung der Standard-Headerdatei `numeric` a) einen `operator+()` deklariert (irgendeinen `operator+()`, egal ob passend oder nicht, zugreifbar oder nicht) oder b) eine andere Standard-Headerdatei einbindet, welche genau dies tut. Im Gegensatz zu Standard-C wird im C++-Standard nicht spezifiziert, welche Standard-Headerdateien sich gegenseitig einbinden, so dass Sie, wenn Sie `numeric` einbinden, unter Umständen auch die Headerdatei `iterator` indirekt einbinden könnten, in der mehrere `operator+()` definiert sind. Ich kenne C++-Compiler, die Beispiel 2 nicht kompilieren, andere, die Beispiel 2 kompilieren, aber ins Stocken kommen, wenn Sie die Zeile »`#include <vector>`« zu Beispiel 2 hinzufügen und so weiter.

Ein bisschen Spaß mit Compilern

Es ist schon schlimm genug, dass der Compiler nicht die richtige Funktion findet, wenn es bereits einen anderen `operator+()` gibt, aber meistens ist der `operator+()`, der in einer Standard-Headerdatei gefunden wird, ein Template und Compiler generieren berüchtigte schwer zu lesende Fehlermeldungen, wenn Templates involviert sind. Zum Beispiel gibt eine populäre Implementierung folgende Fehlermeldungen aus, wenn versucht wird, Beispiel 2 zu kompilieren. (Es sei bemerkt, dass bei dieser Implementierung die Headerdatei `numeric` in der Tat die Headerdatei `iterator` einbindet.)

```
error C2784: 'class std::reverse_iterator<'template-parameter-
1','template-parameter-2','template-parameter-3','template-
parameter-4','template-parameter-5'> __cdecl std::operator
+(template-parameter-5,const class
std::reverse_iterator<'template-parameter-1','template-parameter-
2','template-parameter-3','template-parameter-4','template-
parameter-5'>&)' : could not deduce template argument for
'template-parameter-5' from 'int'
error C2677: binary '+' : no global operator defined which takes
type 'class N::C' (or there is no acceptable conversion)
```

Autsch! Stellen Sie sich vor, wie verwirrt der arme Programmierer sein muss.

- Die erste Fehlermeldung ist nicht lesbar. Der Compiler beschwert sich lediglich (so gut er kann), dass er einen `operator+()` gefunden hat, aber nicht weiß, wie er ihn richtig benutzen kann. Das hilft dem Programmierer aber leider wenig. »Hä?«, fragt er sich am Kopf kratzend, »wann habe ich jemals irgenwo nach einem `reverse_iterator` gefragt?«
- Die zweite Fehlermeldung ist schamlose Lüge und ein Fehler des Compilerherstellers (wenn auch ein verständlicher Fehler, da die Fehlermeldung wahrscheinlich meistens richtig war, bevor Namensräume mehr und mehr benutzt wurden). Sie ist nahe an der richtigen Fehlermeldung »no operator *found* which takes ...«, was dem armen Programmierer aber auch nicht weiterhilft. »Hä?«, fragt sich der Programmierer jetzt empört, »aber da ist doch ein globaler Operator definiert, der ‚class N::C' als Argument hat!«

Wie soll ein normal sterblicher Programmierer jemals herausbekommen, was hier vor sich geht? Und wenn er es erst geschafft hat, wie lautstark wird er den Autor von `N::C` verfluchen? Am besten wäre es gewesen, das Problem von Anfang zu vermeiden, wie wir gleich sehen werden.

Die Lösung

Als wir das Problem des versteckten Namens in bekannter Form mit abgeleiteter und Basisklasse gesehen haben, hatten wir zwei mögliche Lösungen: Entweder der aufrufende Code spezifiziert explizit, welche Funktion aufgerufen werden soll (Beispiel 1b), oder wir schreiben eine `using`-Anweisung an die richtige Stelle, so dass die gewünschte Funktion im richtigen Sichtbarkeitsbereich sichtbar wird (Beispiel 1c). Keine dieser beiden Lösungen funktioniert in diesem Fall. Die erste ist zwar möglich,[11] legt aber dem Programmierer eine nicht akzeptable Last auf die Schultern; die zweite Lösung ist unmöglich.

11. Indem vom Programmierer verlangt wird, dass er die Version von `std::accumulate` benutzt, die ein Prädikat erwartet und explizit jedes Mal sagt, welches benutzt werden soll ... Ein guter Weg Kunden zu verlieren.

Die echte Lösung besteht darin, unseren `operator+()` dorthin zu packen, wo er schon immer wirklich hingehört hat und von Anfang an hätte stehen sollen: in den Namensraum `N`.

```
// Beispiel 2b: Die Lösung
//
// In einer Bibliotheks-Headerdatei:
//
namespace N
{
  class C {};
  int operator+(int i, N::C) { return i+1; }
}
//
// eine main-Funktion zum Ausprobieren:
//
#include <numeric>
int main()
{
  N::C a[10];
  std::accumulate(a, a+10, 0);// ist jetzt ok
}
```

Dieser Code ist portabel und wird von allen standardkonformen Compilern übersetzt, egal was vielleicht schon alles im Namensraum `std` oder einem anderen Namensraum definiert ist. Nun, da der `operator+()` sich im gleichen Namensraum befindet wie sein zweiter Parameter, ist der Compiler in der Lage, wenn er das »+« innerhalb von `std::accumulate` aufzulösen versucht, den richtigen `operator+()` zu sehen aufgrund des Koenig-Lookups. Erinnern Sie sich, dass das Koenig-Lookup besagt, dass der Compiler außer in allen üblichen Sichtbarkeitsbereichen auch die Namensräume durchsuchen soll, aus denen die Parameter der Funktion stammen, um die passende Funktion zu finden. `N::C` befindet sich in dem Namensraum `N`, so dass der Compiler den Namensraum `N` durchsucht und glücklicherweise genau das findet, was er sucht, egal wie viele `operator+()` sonst noch in Reichweite liegen und den Namensraum `std` zumüllen.

Die Schlussfolgerung, die wir ziehen können, ist, dass das Problem deshalb auftrat, weil Beispiel 2 nicht das Schnittstellenprinzip befolgt hat:

Für eine Klasse `X` sind alle Funktionen, inklusive freier Funktionen, die sowohl

- `X` *»aufführen« als auch*
- *zusammen mit `X` »zur Verfügung gestellt« werden,*
  ein logischer Teil von `X`, da sie einen Teil der Schnittstelle von X bilden.

Wenn ein Operation, selbst eine freie Funktion (und insbesondere ein Operator) eine Klasse erwähnt und es beabsichtigt ist, dass sie ein Teil der Schnittstelle der Klasse sein soll, dann sollten Sie immer sicherstellen, dass Sie sie zusammen mit der Klasse zur

Verfügung stellen – was neben anderen Dingen bedeutet, sie in den gleichen Namensraum zu stellen wie die Klasse selbst. Das Problem in Beispiel 2 rührt genau daher, weil wir eine Klasse C geschrieben haben und *einen Teil ihrer Schnittstelle in einen anderen Namensraum gepackt haben.* Sicherzustellen, dass die Klasse und ihre Schnittstelle immer zusammenstehen, ist in jedem Fall immer das Richtige und ein einfacher Weg, um zukünftigen komplizierten Problemen bei der Namenssuche vorzubeugen, wenn andere Leute Ihre Klasse zu benutzen versuchen.

Benutzen Sie Namensräume weise. Packen Sie entweder alles einer Klasse in den gleichen Namensraum – einschließlich der Dinge, die für das unschuldige Auge nicht so aussehen, als ob sie Teil der Klasse sind, wie freie Funktionen, die die Klasse erwähnen (vergessen Sie niemals das Schnittstellenprinzip) – oder packen Sie die Klasse erst gar nicht in einen Namensraum. Die Leute, die Ihren Code benutzen, werden Ihnen dankbar sein.

☑ Richtlinie

Benutzen Sie Namensräume weise. Wenn Sie eine Klasse in einen Namensraum packen, stellen Sie sicher, dass Sie auch alle Hilfsfunktionen und Operatoren in den gleichen Namensraum packen. Sollten Sie dies nicht tun, könnten Sie einige überraschende Effekte in Ihrem Code entdecken.

7 Speicherverwaltung

Lektion 35: Speicherverwaltung – Teil 1 Schwierigkeitsgrad: 3

Wie genau kennen Sie sich mit Speicher aus? Welche verschiedenen Speicherbereiche gibt es? Das folgende Problem deckt die Grundlagen der wichtigsten verschiedenen Speicherbereiche in C++ ab. Das darauffolgende Problem behandelt einige Fragen der Speicherverwaltung noch tiefer.

In C++ gibt es mehrere verschiedene Speicherbereiche, in denen Objekte und Werte, die keine Objekte sind, gespeichert werden können und jeder dieser Bereiche hat andere Merkmale.

Benennen Sie so viele verschiedene Bereiche, wie Sie kennen. Beschreiben Sie für jeden die Performance-Charakteristik und die Lebenszeit der dort gespeicherten Objekte.

Beispiel: Der Stack speichert automatische Variablen, sowohl von den eingebauten Typen als auch von Klassentypen.

Lösung

Die folgende Tabelle fasst die wichtigsten verschiedenen Speicherbereiche von C++ zusammen. Beachten Sie, dass einige der Namen (zum Beispiel »Heap«) nicht als solche im Standard vorkommen; insbesondere »Heap« und »Free Store« sind gebräuchliche und bequeme Begriffe, um zwischen den beiden Arten dynamisch allozierbaren Speichers zu unterscheiden.

Speicherbereich	Charakteristik und Objekt-Lebensdauer
Konstante Daten	Der Speicherbereich für die konstanten Daten speichert String-Literale und andere Daten, deren Wert beim Kompilieren des Programmes bekannt ist. Keine Objekte eines Klassentyps können in diesem Speicherbereich existieren. Alle Daten in diesem Bereich sind während der gesamten Lebenszeit des Programms verfügbar. Weiterhin sind alle Daten nur lesbar und das Verhalten des Programms ist undefiniert, wenn versucht wird, sie zu verändern.

Speicherbereich	Charakteristik und Objekt-Lebensdauer
	Dies ist zum Teil der Fall, da das genaue Speicherformat nicht genau festgelegt ist, sondern beliebigen Optimierungen unterworfen ist. Zum Beispiel könnte ein Compiler sich dafür entscheiden, String-Literale in überlappenden Speicherabschnitten als optionale Optimierung abzulegen.
Stack	Der Stack speichert automatische Variablen. Objekte werden sofort zum Zeitpunkt ihrer Definition konstruiert und sofort zerstört, wenn ihr Gültigkeitsbereich verlassen wird, so dass es für einen Programmierer keine Möglichkeit gibt, allozierten, aber noch nicht initialisierten Speicherplatz auf dem Stack direkt zu manipulieren (außer wenn willentlich mit expliziten Destruktor-Aufrufen und Placement-`new` rumgespielt wird).
	Speicherallozierung auf dem Stack ist typischerweise viel schneller als dynamische Speicherallozierung (auf dem Heap oder freien Speicher), da die Allozierung von Speicherplatz auf dem Stack nur die Erhöhung eines Stackzeigers erfordert und keine komplexe Speicherverwaltung.
Free Store	Der Free Store ist einer der beiden Speicherbereiche für dynamisch allozierten Speicher, der via `new` alloziert und mit `delete` wieder freigegeben wird.
	Die Lebensdauer eines Objektes kann kürzer sein als die Zeit, in der der Speicher alloziert ist. Das heißt im Free Store kann Speicherplatz alloziert werden, ohne dass dieser sofort initialisiert wird, und Objekte können zerstört werden, ohne dass der Speicherplatz sofort dealloziert wird. Während der Zeitspanne, in der der Speicherplatz alloziert ist, aber außerhalb der Lebensdauer des Objekts, kann auf den Speicherplatz über einen `void*`-Zeiger zugegriffen werden, es darf jedoch auf keine der Elementfunktionen zugegriffen, ihre Adresse ermittelt oder sonstiges getan werden.
Heap	Der Heap ist der andere Speicherbereich für dynamisch allozierten Speicher, der durch die `malloc()`-Funktion und ihre Varianten belegt und mittels `free()` wieder freigegeben wird.
	Beachten Sie, auch wenn die globalen Default-Operatoren `new` und `delete` von einem bestimmten Compiler mittels `malloc()` und `free()` implementiert sein könnten, ist der Heap trotzdem nicht das Gleiche wie der Free Store und Speicher, der in dem einen Speicherbereich alloziert wurde, kann nicht sicher in dem anderen freigegeben werden.
	Speicher, der über den Heap alloziert wurde, kann für Objekte eines Klassentyps durch Konstruktion mittels Placement-`new` und expliziter Destruktion benutzt werden. Ist dies der Fall, so gelten auch hier die Bemerkungen zur Lebendauer von Objekten im Free Store.

Speicherbereich	Charakteristik und Objekt-Lebensdauer
Global/Static	Für globale oder statische Variablen und Objekte wird der Speicherplatz beim Programmstart alloziert, aber unter Umständen nicht initialisiert, bis das Programm mit seiner Ausführung begonnen hat. Zum Beispiel wird eine statische Variable einer Funktion erst initialisiert, wenn das Programm die Funktion zum ersten Mal aufruft. Die Reihenfolge der Initialisierung globaler Variablen über mehrere Übersetzungseinheiten hinweg ist nicht definiert, was besondere Sorgfalt erfordert, wenn Abhängigkeiten zwischen globalen Objekten gemeistert werden müssen (einschließlich statischer Klassenvaribalen). Wie immer kann auf nicht initialisierten Speicher über einen `void*`-Zeiger zugegriffen werden, es dürfen aber außerhalb der Lebendauer des jeweiligen Objekts keine Elementfunktionen aufgerufen oder referenziert werden.

Es ist wichtig, zwischen dem »Heap« und dem »Free Store« zu unterscheiden, da der Standard absichtlich die Frage unspezifiziert lässt, ob diese beiden Bereiche zusammenhängen. Wenn zum Beispiel Speicher via `::operator delete()` freigegeben wird, so besagt die abschließende Notiz in Abschnitt 18.4.1.1 des C++-Standards Folgendes:

> *»It is unspecified under what conditions part or all of such reclaimed storage is allocated by a subsequent call to* `operator new` *or any of* `calloc`, `malloc`, *or* `realloc`, *declared in* `<cstdlib>`.«

Weiterhin ist auch nicht spezifiziert, ob `new` und `delete` mittels `malloc` und `free` implementiert sind. Es ist jedoch spezifiziert, dass `malloc` und `free` *nicht* mit Hilfe von `new` und `delete` implementiert sein dürfen, entsprechend dem Abschnitt 20.4.6, Absätze 3 und 4:

> *»The functions* `calloc()`, `malloc()`, *and* `realloc()` *do not attempt to allocate storage by calling* `::operator new()`.«

> *»The function* `free()` *does not attempt to deallocate storage by calling* `::operator delete()`.«

Effektiv verhalten sich Heap und Free Store unterschiedlich und es wird unterschiedlich darauf zugegriffen, weshalb Sie sicherstellen sollten, dass Sie sie auch unterschiedlich benutzen.

☑ Richtlinie

Verstehen Sie die fünf wichtigsten verschiedenen Speicherbereiche in C++, worin Sie sich unterscheiden und wie sie sich verhalten: Stack (automatische Variablen); Free Store (`new`/`delete`); Heap(`malloc`/`free`); globaler Bereich (statische und globale Variablen und so weiter); konstanter Bereich (String-Literale und so weiter).

☑ Richtlinie

Bevorzugen Sie den Free Store (`new`/`delete`). Vermeiden Sie die Verwendung des Heaps (`malloc`/`free`).

Lektion 36: Speicherverwaltung – Teil 2 Schwierigkeitsgrad: 6

Denken Sie darüber nach, Ihre eigene klassenspezifische Speicherverwaltung zu schreiben oder gar die globalen C++-Operatoren `new` und `delete` zu ersetzen? Versuchen Sie sich erst mal an folgendem Problem.

Der folgende Code zeigt Klassen, die ihre eigene Speicherverwaltung durchführen. Finden Sie so viele Speicherverwaltungsfehler wie möglich und beantworten Sie die zusätzlichen Fragen.

1. Betrachten Sie den folgenden Code:

```
class B
{
public:
  virtual ~B();
  void operator delete  ( void*, size_t ) throw();
  void operator delete[]( void*, size_t ) throw();
  void f( void*, size_t ) throw();
};
class D : public B
{
public:
  void operator delete  ( void* ) throw();
  void operator delete[]( void* ) throw();
};
```

   Warum haben `B`'s `delete`-Operatoren einen zweiten Parameter, die von `D` hingegen nicht? Sehen Sie einen Weg die Funktionsdeklarationen zu verbessern?

2. Fortsetzend mit obigen Code: Welcher `operator delete()` wird bei jeder der folgenden `delete`-Anweisungen aufgerufen? Warum und mit welchen Parametern?

```
D* pd1 = new D;
delete pd1;
B* pb1 = new D;
delete pb1;
D* pd2 = new D[10];
delete[] pd2;
B* pb2 = new D[10];
delete[] pb2;
```

3. Sind die beiden folgenden Zuweisungen erlaubt?

```
B b;
typedef void (B::*PMF)(void*, size_t);
PMF p1 = &B::f;
PMF p2 = &B::operator delete;
```

4. Gibt es irgendwelche speicherverwaltungs-bezogene Fehler oder Sachen im folgenden Code:

```
class X
{
public:
  void* operator new( size_t s, int )
                  throw( bad_alloc )
  {
    return ::operator new( s );
  }
};

class SharedMemory
{
public:
  static void* Allocate( size_t s )
  {
    return OsSpecificSharedMemAllocation( s );
  }
  static void  Deallocate( void* p, int i )
  {
    OsSpecificSharedMemDeallocation( p, i );
  }
};

class Y
{
public:
  void* operator new( size_t s,
                      SharedMemory& m ) throw( bad_alloc )
  {
    return m.Allocate( s );
  }
  void  operator delete( void* p,
                         SharedMemory& m,
                         int i ) throw()
  {
     m.Deallocate( p, i );
  }
};
```

```
void operator delete( void* p ) throw()
{
  SharedMemory::Deallocate( p );
}

void operator delete( void* p,
                      std::nothrow_t& ) throw()
{
  SharedMemory::Deallocate( p );
}
```

Lösung

Lassen Sie uns die Frage eine nach der anderen durchgehen.

1. Betrachten Sie den folgenden Code:

```
class B
{
public:
  virtual ~B();
  void operator delete  ( void*, size_t ) throw();
  void operator delete[]( void*, size_t ) throw();
  void f( void*, size_t ) throw();
};
class D : public B
{
public:
  void operator delete  ( void* ) throw();
  void operator delete[]( void* ) throw();
};
```

   Warum haben `B`'s `delete`-Operatoren einen zweiten Parameter, die von `D` hingegen nicht?

   Die Antwort auf diese Frage ist: Es handelt sich lediglich um eine persönliche Vorliebe. Beides sind gewöhnliche Deallozierungs-Funktionen und keine Placement-`delete`s (siehe Abschnitt 3.7.3.2/2 im C++-Standard).

   Wie dem auch sei, hier lauert auch ein Speicherverwaltungsfehler im Dickicht. Beide Klassen stellen einen `operator delete()` und einen `operator delete[]()` zur Verfügung, jedoch ohne die korrespondierenden `operator new()` und `operator new[]()`. Dies ist extrem gefährlich, da die Standard-Operatoren `operator new()` und `operator new[]()` wahrscheinlich nicht das Richtige tun. (Betrachten Sie zum Beispiel den Fall, wenn eine weitere abgeleitete Klasse eigene Versionen von `operator new()` oder `operator new[]()` definiert.)

☑ Richtlinie

Stellen Sie immer sowohl ein klassenspezifisches `new` (oder `new[]`) als auch ein klassenspezifisches `delete` (oder `delete[]`) zur Verfügung, wenn Sie eines davon definieren.

Und nun zum zweiten Teil der Frage: Sehen Sie einen Weg die Funktionsdeklarationen zu verbessern?

Alle Versionen von `operator new()` und `operator delete()` sind immer statische Funktionen, selbst wenn sie nicht als `static` deklariert sind. Obwohl Sie C++ nicht zwingt, explizit »`static`« zu schreiben, wenn Sie Ihre eigenen Versionen von `new` oder `delete` deklarieren, ist es besser, es trotzdem zu tun, da Sie so eine Gedächtnisstütze für sich selbst erhalten, während Sie den zugehörigen Code schreiben, und es ist auch eine Gedächtnisstütze für den nächsten Programmierer, der Ihren Code warten muss.

☑ Richtlinie

Deklarieren Sie `operator new()` und `operator delete()` immer explizit als statische Funktionen. Sie können niemals nicht-statische Elementfunktionen sein.

2. Fortsetzend mit obigen Code: Welcher `operator delete()` wird bei jeder der folgenden `delete`-Anweisungen aufgerufen? Warum und mit welchen Parametern?

```
D* pd1 = new D;
delete pd1;
```

Hierdurch wird `D::operator delete(void*)` aufgerufen.

```
B* pb1 = new D;
delete pb1;
```

Hierdurch wird ebenfalls `D::operator delete(void*)` aufgerufen. Da der Destruktor von `B` virtuell ist, wird natürlich `D`'s Destruktor ordnungsgemäß aufgerufen, aber die Tatsache, dass `B`'s Destruktor virtuell ist, bedeutet auch implizit, dass `D::operator delete()` aufgerufen wird, obwohl `B::operator delete()` nicht virtuell ist (was er auch gar nicht sein kann).

Als Randbemerkung für die, die sich dafür interessieren, wie Compiler solche Dinge implementieren: Die gebräuchliche Vorgehensweise ist, dass dem Code, der wirklich für jeden Destruktor generiert wird, ein unsichtbares »Soll-das-Objekt-nach-seiner-Zerstörung-gelöscht-werden«-Flag hinzugefügt wird, das entsprechend gesetzt wird (`false`, wenn ein automatisches Objekt zerstört wird, `true`, wenn ein dynamisches Objekt zerstört wird). Die letzte Aktion, die der generierte Destruktor-Code ausführt, ist, dieses Flag zu überprüfen und, wenn es `true` ist, den

richtigen `operator delete()` aufzurufen.[1] Diese Technik stellt automatisch das korrekte Verhalten sicher, nämlich, dass der `operator delete()` sich scheinbar wie eine »virtuelle« Funktion benimmt, obwohl es sich um eine statische Funktion handelt, die deshalb gar nicht virtuell sein kann.

```
D* pd2 = new D[10];
delete[] pd2;
```

Hierdurch wird `D::operator delete[](void*)` aufgerufen.

```
B* pb2 = new D[10];
delete[] pb2;
```

Hier ist das Verhalten nicht definiert. Die C++-Sprache erfordert, dass der statische Typ eines Zeigers, der an den `operator delete[]()` übergeben wird, der gleiche Typ sein muss wie der dynamische Typ des Zeigers. Für ausführlichere Informationen zu diesem Thema siehe auch Scott Meyers »Never Treat Arrays Polymorphically« in Meyers99.

☑ Richtlinie

Behandeln Sie Arrays niemals polymorph.

☑ Richtlinie

Bevorzugen Sie `vector<>` oder `deque<>` anstelle von Arrays.

3. Sind die beiden folgenden Zuweisungen erlaubt?

```
B b;
typedef void (B::*PMF)(void*, size_t);
PMF p1 = &B::f;
PMF p2 = &B::operator delete;
```

Die erste Zuweisung ist in Ordnung; wir weisen einfach die Adresse einer Elementfunktion einem Zeiger auf eine Elementfunktion zu.

Die zweite Zuweisung ist jedoch illegal, da `void operator delete( void*, size_t ) throw()` *keine* nicht-statische Elementfunktion von `B` ist, selbst wenn sie so, wie sie niedergeschrieben ist, so aussehen mag. Erinnern Sie sich daran, dass `operator new()` und `operator delete()` immer statische Elementfunktionen sind, auch wenn sie nicht explizit als `static` deklariert werden. Es ist eine gute Angewohnheit, sie

1. Die korrekte Variante dieser Technik für den `operator delete[]()` für Arrays sei dem Leser als Übung überlassen.

immer als statisch zu deklarieren, nur um sicherzustellen, dass diese Tatsache allen Programmierern bewusst wird, wenn diese Ihren Code lesen.

☑ Richtlinie

Deklarieren Sie `operator new()` *und* `operator delete()` *immer explizit als statische Funktionen. Sie können niemals nicht-statische Elementfunktionen sein.*

4. Gibt es irgendwelche speicherverwaltungs-bezogene Fehler oder Sachen im folgenden Code:

```
class X
{
public:
  void* operator new( size_t s, int )
                  throw( bad_alloc )
  {
    return ::operator new( s );
  }
};
```

Hierdurch wird ein Speicherloch heraufbeschworen, da kein korrespondierendes Placement-`delete` existiert. Ähnlich weiter unten:

```
class SharedMemory
{
public:
  static void* Allocate( size_t s )
  {
    return OsSpecificSharedMemAllocation( s );
  }
  static void  Deallocate( void* p, int i )
  {
    OsSpecificSharedMemDeallocation( p, i );
  }
};

class Y
{
public:
  void* operator new( size_t s,
                      SharedMemory& m ) throw( bad_alloc )
  {
    return m.Allocate( s );
  }
```

Auch hierdurch wird ein Speicherloch heraufbeschworen, da kein `operator delete()` mit der gleichen Signatur existiert. Wenn eine Exception während der Konstruktion eines Objekts, das sich in dem von dieser Funktion allozierten Speicher befindet, geworfen wird, so wird dieser Speicher nicht ordentlich wieder freigegeben. Betrachten Sie zum Beispiel folgenden Code:

```
SharedMemory shared;
...
new (shared) T; // wenn T::T() eine Exception wirft,
                // entsteht ein Speicherloch
```

Außerdem kann aller Speicher, der von diesem `operator new()` alloziert wird, nicht sicher gelöscht werden, da die Klasse keinen gewöhnlichen `operator delete()` zur Verfügung stellt. Das bedeutet, dass der `operator delete()` einer Basisklasse oder einer abgeleiteten Klasse oder vielleicht auch der globalen sich um die Deallozierung kümmern muss (was mit großer Sicherheit fehlschlagen wird, wenn Sie nicht alle umgebenden `delete`-Operatoren ersetzt haben, was lästig und bösartig wäre).

```
  void  operator delete( void* p,
                         SharedMemory& m,
                         int i ) throw()
  {
     m.Deallocate( p, i );
  }
};
```

Dieser `Y::operator delete()` ist nutzlos, da er niemals aufgerufen werden kann.

```
void operator delete( void* p ) throw()
{
  SharedMemory::Deallocate( p );
}
```

Dies ist ein ernsthafter Fehler, da dieser `operator delete()`, der den globalen Default-Operator ersetzt, Speicher freizugeben versuchen wird, der ganz normal mit dem Default-Operator `::operator new()` alloziert wurde und nicht mit `SharedMemory::Allocate()`. Das Beste, worauf Sie dabei hoffen können, ist ein schneller Core-Dump. Bösartig.

```
void operator delete( void* p,
                      std::nothrow_t& ) throw()
{
  SharedMemory::Deallocate( p );
}
```

Die gleichen Bemerkungen wie oben gelten auch hier, nur dass es diesmal etwas subtiler ist. Dieser ersetzende `operator delete()` wird nur dann aufgerufen, wenn ein Ausdruck wie »`new (nothrow) T`« fehlschlägt, da der Konstruktor mit einer

Exception abbricht, wodurch wiederum versucht wird Speicher freizugeben, der nicht mittels `SharedMemory::Allocate()` alloziert wurde. Bösartig und heimtückisch.

☑ Richtlinie

Stellen Sie immer sowohl ein klassenspezifisches `new` (oder `new[]`) als auch ein klassenspezifisches `delete` (oder `delete[]`) zur Verfügung, wenn Sie eines davon definieren.

Wenn Sie alle diese Antworten gefunden und sie auch alle verstanden haben, sind Sie definitiv auf dem richtigen Weg ein Experte für Speicherverwaltung zu werden.

Lektion 37: `auto_ptr` Schwierigkeitsgrad: 8

In dieser Lektion geht es darum, wie Sie den Standard-`auto_ptr` sicher und effektiv einsetzen können.

Historische Randbemerkung: Die ursprüngliche, einfachere Version dieser Lektion – erschienen als Sonderausgabe des *Guru of the Week* – wurde erstmals anlässlich der Abwahl des *Final Draft International Standard for Programming Language C++* veröffentlicht. Es war bekannt/vermutet, dass `auto_ptr` sich im letzten Meeting, bei dem der Standard als vollständig verabschiedet wurde (Morristown, New Jersey, November 1997), ein letztes Mal ändern würde, so dass das folgende Problem einen Tag, bevor das Meeting begann, aufgebracht wurde. Die Lösung, frisch aktualisiert, um die Änderungen am Standard des Vortages zu reflektieren, wurde die erste öffentliche Abhandlung zum Standard-`auto_ptr`.

Vielen Dank von uns allen gebührt Bill Gibbons, Greg Colvin, Steve Rumsby und anderen, die hart an der Verbesserung des endgültigen `auto_ptr` gearbeitet haben. Insbesondere Greg hat an `auto_ptr` und verwandten Smart-Pointer-Klassen für viele Jahre gearbeitet, um die verschiedenen Sorgen und Anforderungen des Komitees zu befriedigen, und verdient dafür öffentliche Anerkennung.

Dieses Problem, nun mit einer bemerkenswert umfassenderen und verbesserten Lösung, illustriert die Gründe für die Änderungen, die kurz vor zwölf noch vorgenommen wurden, und zeigt außerdem, wie Sie am besten Gebrauch von `auto_ptr` machen können.

Kommentieren Sie den folgenden Code: Was ist gut, was ist sicher, was ist erlaubt und was nicht?

```
auto_ptr<T> source()
{
  return auto_ptr<T>( new T(1) );
}
```

```
void sink( auto_ptr<T> pt ) { }
void f()
{
  auto_ptr<T> a( source() );
  sink( source() );
  sink( auto_ptr<T>( new T(1) ) );
  vector< auto_ptr<T> > v;
  v.push_back( auto_ptr<T>( new T(3) ) );
  v.push_back( auto_ptr<T>( new T(4) ) );
  v.push_back( auto_ptr<T>( new T(1) ) );
  v.push_back( a );
  v.push_back( auto_ptr<T>( new T(2) ) );
  sort( v.begin(), v.end() );
  cout << a->Value();
}
class C
{
public:    /*...*/
protected: /*...*/
private:   /*...*/
  auto_ptr<CImpl> pimpl_;
};
```


Lösung

Die meisten Leute haben bereits davon gehört, dass der Standard ein Smart-Pointer-Template names `auto_ptr` enthält, aber nicht jeder benutzt es täglich. Das ist schade, denn es ist so, dass `auto_ptr` eine elegante Lösung für bekannte Programmierprobleme in C++ darstellt, und der gekonnte Einsatz von `auto_ptr` führt auch zu robusterem Code. Diese Lektion zeigt, wie Sie `auto_ptr` korrekt benutzen, um Ihren Code sicherer zu machen – und wie Sie die gefährlichen, aber verbreiteten Missbräuche von `auto_ptr` vermeiden, die zu periodischen und schwer zu findenden Fehlern führen.

Warum »Auto«-Zeiger?

`auto_ptr` ist nur einer eines weiten Felds so genannter Smart-Pointers. Viele kommerzielle Bibliotheken stellen noch ausgeklügeltere Arten von Smart-Pointern zur Verfügung, die ganz wilde und wundervolle Sachen können, angefangen bei der Verwaltung von Referenzzählern bis zur Unterstützung fortgeschrittener Proxy-Dienste. Stellen Sie sich `auto_ptr` als Ford Escort unter den Smart-Pointern vor – ein einfacher Smart-Pointer für den allgemeinen Gebrauch, der nicht die ganzen Extras und den Luxus von speziellen Smart-Pointern oder sehr leistungsstarken Smart-Pointern hat, aber viele Dinge gut macht und optimal für eine regelmäßige Verwendung geeignet ist.

Was `auto_ptr` tut, ist, dynamisch allozierte Objekte zu besitzen und eine automatisches Clean-Up durchzuführen, wenn das Objekt nicht länger benötigt wird. Hier ein einfaches Beispiel unsicheren Codes ohne `auto_ptr`:

```
// Beispiel 1(a): Ursprünglicher Code
//
void f()
{
  T* pt( new T );
  /*...mehr Code...*/
  delete pt;
}
```

Viele von uns schreiben solchen Code jeden Tag. Wenn `f()` nur ein Dreizeiler ist, der nichts Außergewöhnliches tut, ist das unter Umständen auch in Ordnung. Wenn f() jedoch nie zu der Ausführung der `delete`-Anweisung kommt, entweder durch ein vorzeitiges `return` oder dadurch, dass innerhalb der Funktion eine Exception geworfen wird, wird das allozierte Objekt nicht gelöscht und wir haben ein klassisches Speicherloch.

Ein einfacher Weg, um Beispiel 1(a) sicher zu machen, besteht darin, den Zeiger in eine »klügere« zeiger-ähnliche Klasse zu packen, die den Zeiger besitzt und die, wenn sie zerstört wird, das Objekt, auf das der Zeiger zeigt, automatisch löscht. Da ein solcher Smart-Pointer einfach als automatisches Objekt (also ein Objekt auf dem Stack, das automatisch zerstört wird, wenn sein Gültigkeitsbereich verlassen wird) genutzt wird, wurde er »Auto-Pointer« getauft:

```
// Beispiel 1(b): Sicherer Code mit auto_ptr
//
void f()
{
  auto_ptr<T> pt( new T );
  /*...mehr Code...*/
} // cool: pt's Destruktor wird automatisch aufgerufen,
  // wenn pt's Gültigkeitsbereich verlassen wird,
  // und das T-Objekt automatisch gelöscht
```

Nun wird niemals ein Speicherloch durch das T-Objekt hervorgerufen, egal ob die Funktion normal durchlaufen oder durch eine Exception abgebrochen wird, da `pt`'s Destruktor immer aufgerufen wird, wenn der Stack aufgeräumt wird, wird auch automatisch das Clean-Up ausgeführt.

Letztendlich ist die Benutzung eines `auto_ptr`-Objekts genauso einfach wie bei einem eingebauten Zeiger, und um die Ressource »zurückzufordern«, um selbst den Besitz zu übernehmen, können wir einfach `release()` aufrufen.

```
// Beispiel 2: Benutzung eines auto_ptr
//
void g()
```

```
{
  T* pt1 = new T;
  // jetzt besitzen wir das allozierte Objekt
  // wir geben den Besitz an auto_ptr ab
  auto_ptr<T> pt2( pt1 );
  // wir benutzen auto_ptr ganz genauso wie wir
  // einen einfachen Zeiger benutzen würden
  *pt2 = 12;       // genauso wie "*pt1 = 12;"
  pt2->SomeFunc(); // genauso wie "pt1->SomeFunc();"
  // mit get() ermittelt man den Wert des Zeigers
  assert( pt1 == pt2.get() );
  // mit release() erlangen wir den Besitz zurück
  T* pt3 = pt2.release();
  // wir löschen das Objekt selbst, da es jetzt
  // von keinem auto_ptr mehr besessen wird
  delete pt3;
} // pt2 besitzt keinen Zeiger mehr und wird deshalb
  // auch nicht versuchen delete aufzurufen
```

Schließlich können wir die `auto_ptr`-Funktion `reset()` benutzen, um dem `auto_ptr` ein anderes Objekt zuzuweisen. Wenn der `auto_ptr` bereits ein Objekt besitzt, wird dieses zunächst gelöscht, so dass der Aufruf von `reset()` eigentlich nichts anderes bewirkt, als den `auto_ptr` zu zerstören und einen neuen zu erzeugen, der das neue Objekt besitzt.

```
// Beispiel 3: Benutzung von reset()
//
void h()
{
  auto_ptr<T> pt( new T(1) );
  pt.reset( new T(2) );
    // löscht das erste T-Objekt, das
    // mit "new T(1)" alloziert wurde
} // pt verlässt seinen Gültigkeitsbereich
  // und das zweite T-Objekt wird auch gelöscht
```

Einpacken von Zeiger-Elementvariablen

Ähnlich kann `auto_ptr` benutzt werden, um sicher Zieger-Elementvariablen einzupacken. Betrachten Sie das folgende typische Beispiel für das Pimpl- (oder Compiler-Firewall) -Idiom.[2]

```
// Beispiel 4(a): Eine typische Pimpl-Implementierung
//
// Datei c.h
//
```

2. Das Pimpl-Idiom ist nützlich zum Reduzieren der Übersetzungszeiten umfangreicher Projekte, da es umfassender Neukompilation von Client-Code vorbeugt, wenn sich private Details der Klasse `C` ändern. Für ausführlichere Informationen zum Pimpl-Idiom siehe Lektionen 26 bis 30.

```
class C
{
public:
  C();
  ~C();
  /*...*/
private:
  class CImpl; // Vorwärtsdeklaration
  CImpl* pimpl_;
};
//
// Datei c.cpp
//
class C::CImpl { /*...*/ };
C::C() : pimpl_( new CImpl ) { }
C::~C() { delete pimpl_; }
```

Kurzgefasst werden die privaten Details von `C` abgespalten und in ein separates Implementierungsobjekt gepackt, das hinter einem undurchsichtigen Zeiger versteckt wird. Dabei ist `C`'s Konstruktor dafür verantwortlich, das private Pimpl-Hilfsobjekt zu allozieren, welches die versteckten Klasseninterna enthält, und der Destruktor von `C` ist dafür verantwortlich, das Pimpl-Objekt wieder zu löschen. Durch die Benutzung von `auto_ptr` können wir uns das Leben aber einfacher machen:

```
// Beispiel 4(b): Eine sicherere Pimpl-Implementierung,
//                die auto_ptr benutzt
//
// Datei c.h
//
class C
{
public:
  C();
  ~C();
  /*...*/
private:
  class CImpl; // Vorwärtsdeklaration
  auto_ptr<CImpl> pimpl_;
  C& operator = ( const C& );
  C( const C& );
};
//
// Datei c.cpp
//
class C::CImpl { /*...*/ };
C::C() : pimpl_( new CImpl ) { }
C::~C() {}
```

Nun muss sich der Destruktor nicht mehr um das Löschen des `pimpl_`-Zeigers kümmern, da dies der `auto_ptr` automatisch übernimmt. Dazu kommt, dass `C::C()` weniger

Arbeit investieren muss, um Fehler bei der Konstruktion des Objekts zu entdecken und zu behandeln, da `pimpl_` immer automatisch dem korrektem Clean-Up unterzogen wird. Dies ist einfacher, als Zeiger manuell zu verwalten, und folgt der guten Praxis, den Besitz von Ressourcen in Objekte zu kapseln – eine Aufgabe, für die `auto_ptr` ideal geeignet ist. Wir werden uns diesem Beispiel am Ende dieser Lektion noch einmal zuwenden.

Besitz, Quellen und Senken

Das ist schon alles ganz hübsch für sich selbst genommen, aber es wird noch besser. Es ist auch sehr nützlich, `auto_ptr`-Objekte an Funktionen zu übergeben oder von ihnen zurückzubekommen – als Funktionsparameter oder Rückgabewert.

Um zu sehen warum, betrachten wir als Erstes, was passiert, wenn ein `auto_ptr` kopiert wird: Ein `auto_ptr` besitzt das Objekt, auf das der in `auto_ptr` enthaltene Zeiger zeigt, und nur ein `auto_ptr` kann ein Objekt zu einem Zeitpunkt besitzen. Wenn Sie einen `auto_ptr` kopieren, übertragen Sie gleichzeitig den Besitz vom Quell-`auto_ptr` auf den Ziel-`auto_ptr`; wenn der Ziel-`auto_ptr` bereits ein Objekt besitzt, wird dieses zuvor gelöscht. Nach dem Kopieren besitzt nur noch der Ziel-`auto_ptr` das Objekt und wird es zu gegebener Zeit löschen, während der Quell-`auto_ptr` in den `null`-Zustand zurückversetzt wird und nicht länger benutzt werden kann, um sich auf das besessene Objekt zu beziehen.

Zum Beispiel:

```
// Beispiel 5: Übertragen des Besitzes von
// /            einem auto_ptr auf einen anderen
//
void f()
{
  auto_ptr<T> pt1( new T );
  auto_ptr<T> pt2;
  pt1->DoSomething(); // OK
  pt2 = pt1;  // nun besitzt pt2 den Zeiger
              // und pt1 nicht mehr
  pt2->DoSomething(); // OK
} // beim Verlassen des Gültigkeitsbereichs wird
  // der Zeiger durch pt2's Destruktor gelöscht,
  // der Destruktor von pt1 bewirkt nichts
```

Seien Sie jedoch vorsichtig nicht in die Falle zu tappen, zu versuchen, einen `auto_ptr` zu benutzen, der nichts mehr besitzt:

```
// Beispiel 6: Versuchen Sie nicht einen auto_ptr
//             zu benutzen, der nichts besitzt
//
```

```
void f()
{
  auto_ptr<T> pt1( new T );
  auto_ptr<T> pt2;
  pt2 = pt1;  // nun besitzt pt2 den Zeiger,
              // und pt1 nicht mehr
  pt1->DoSomething();
              // Fehler: Zugriff über Null-Pointer
}
```

Mit diesem Wissen im Hinterkopf fangen wir an zu sehen, wie gut `auto_ptr` mit Quellen und Senken zusammenarbeitet. Eine »Quelle« ist eine Funktion oder eine andere Operation, die eine neue Ressource erzeugt und dann typischerweise diese abtritt und den Besitz der Ressource aufgibt. Eine »Senke« ist eine Funktion, die genau das Gegenteil macht – nämlich den Besitz eines existierenden Objekts zu übernehmen (und es typischerweise zu zerstören). Aber anstatt, dass Quellen und Senken nur einfache Zeiger zurückgeben oder erwarten, ist es meistens besser, gleich einen Smart-Pointer zurückzugeben oder zu erwarten, der die Ressource besitzt.

Dies führt uns zum ersten Teil des Codes für diese Lektion:

```
auto_ptr<T> source()
{
  return auto_ptr<T>( new T(1) );
}
void sink( auto_ptr<T> pt ) { }
```

Dies ist sowohl erlaubt als auch sicher. Beachten Sie die Eleganz dessen, was hier passiert:

1. `source()` alloziert ein neues Objekt und gibt es an den Aufrufer auf einem absolut sicheren Wege zurück, indem es dem Aufrufer den Besitz des Zeigers überträgt. Selbst wenn der Aufrufer den Rückgabewert ignoriert (natürlich würden Sie nie Code schreiben, der Rückgabewerte ignoriert, oder?), wird das allozierte Objekt immer sicher gelöscht.

   Siehe auch Lektion 19, warum dies ein wichtiges Idiom ist, da die Rückgabe eines Ergebnisses eingepackt in etwas wie `auto_ptr` manchmal der einzige Weg ist, eine Funktion wirklich exception-sicher zu machen.

2. `sink()` erwartet einen `auto_ptr`, der via Wert übergeben wird, wodurch der Besitz auf diesen Parameter übergeht. Wenn `sink()` fertig ist, wird der Zeiger gelöscht, da das lokale `auto_ptr`-Objekt seinen Gültigkeitsbereich verlässt (falls `sink()` nicht selbst den Besitz an jemand anderen übergeben hat). Da die `sink()`-Funktion so wie oben geschrieben praktisch nichts mit ihrem Parameter tut, ist der Aufruf »`sink(pt);`« nichts weiter als ein fantasievoller Weg »`pt.reset(0);`« zu schreiben.

Der nächste Code-Abschnitt zeigt `source()` und `sink()` in Aktion.

```
void f()
{
  auto_ptr<T> a( source() );
```

Dies ist wiederum sowohl erlaubt als auch sicher. Hier übernimmt `f()` den Besitz des Zeigers, der von `source()` zurückgeliefert wird, und dieser wird (wenn man von einigen späteren Problemen in `f()` absieht) automatisch gelöscht werden, wenn die automatische Variable ihren Gültigkeitsbereich verlässt. Dies ist schön und genauso, wie die Rückgabe eines `auto_ptr` via Wert funktionieren soll.

```
sink( source() );
```

Und wieder ist dies erlaubt und sicher. Mit den trivialen (soll heißen »leeren«) Definitionen von `source()` und `sink()` in unserem Beispiel ist dies nur ein fantasievoller Weg, um »`delete new T(1);`« zu schreiben. Ist dies wirklich sinnvoll? Nun, wenn Sie sich `source()` als eine nicht-triviale Factory-Funktion vorstellen und `sink()` als eine nicht-triviale Consumer-Funktion, dann ja, dann macht es eine Menge Sinn und lässt sich regelmäßig in echten Programmen wiederfinden.

```
sink( auto_ptr<T>( new T(1) ) );
```

Immer noch erlaubt und sicher. Ein weiterer fantasievoller Weg »`delete new T(1);`« zu schreiben, und es ist ein nützliches Idiom, wenn `sink()` eine nicht-triviale Consumer-Funktion ist, die den Besitz des Objektes übernimmt, auf das gezeigt wird.

Seien Sie jedoch auf der Hut: Benutzen Sie `auto_ptr` niemals anders als auf den Wegen, die ich soeben beschrieben habe. Ich habe viele Programmierer gesehen, die versucht haben, `auto_ptr` auf andere Art und Weise zu benutzen, so wie sie auch andere Objekte benutzen. Das Problem hierbei ist, das `auto_ptr`-Objekte sich fast immer *nicht* wie andere Objekte verhalten. Hier die grundlegende Schwierigkeit, und ich stelle sie heraus, um sicherzustellen, dass sie sich hervorhebt:

Für `auto_ptr` *sind Kopien NICHT äquivalent.*

Es stellt sich heraus, dass dies zu bedeutenden Effekten führt, wenn Sie versuchen `auto_ptr` mit generischem Code zu verwenden, der Kopien anlegt und nicht unbedingt darauf vorbereitet ist, dass Kopien nicht äquivalent sein könnten (schließlich sind sie es gewöhnlich). Betrachten Sie den folgenden Code, der immer wieder in der C++-Newsgroup auftaucht:

```
vector< auto_ptr<T> > v;
```

Dies ist nicht erlaubt und ist ganz bestimmt nicht sicher! Seien Sie auf der Hut. Es ist *niemals* sicher, `auto_ptr` in Standard-Container zu packen. Einige Leute werden Ihnen sagen, dass ihre Compiler und Bibliotheken das problemlos übersetzen, und andere werden Ihnen erzählen, dass sie genau so etwas in der Dokumentation eines bestimmten populären Compilers empfohlen gesehen haben. Hören Sie nicht auf sie.

Das Problem ist, dass `auto_ptr` nicht die Anforderungen für einen Typ erfüllt, der in Standard-Container gepackt werden kann, da Kopien von auto_ptr nicht äquivalent sind. Zum einen spricht nichts dagegen, dass ein `vector` sich dafür entscheiden kann, eine »zusätzliche« interne Kopie von Objekten anzulegen, die er enthält. Aber warten Sie, es kommt noch dicker:

```
v.push_back( auto_ptr<T>( new T(3) ) );
v.push_back( auto_ptr<T>( new T(4) ) );
v.push_back( auto_ptr<T>( new T(1) ) );
v.push_back( a );
```

(Beachten Sie, dass das Kopieren von `a` in `v` bedeutet, dass das Objekt `a` nicht länger den Zeiger besitzt, den es am Anfang zugewiesen bekommen hat. Mehr dazu gleich.)

```
v.push_back( auto_ptr<T>( new T(2) ) );
sort( v.begin(), v.end() );
```

Illegal. Unsicher. Wenn Sie generische Funktionen wie `sort()` aufrufen, die Elemente kopieren, müssen die Funktionen davon ausgehen können, dass die Kopien äquivalent sind. Wenigstens eine verbreitete `sort()`-Funktion legt intern eine Kopie eines »Pivot«-Elements an, und wenn Sie versuchen, dies mit `auto_ptr`-Objekten zu tun, wird einfach eine Kopie des Pivot-`auto_ptr` angelegt (wobei der Besitz übernommen wird, den eine temporäre `auto_ptr`-Variable bekommt), die die restliche Arbeit an der Sequenz verrichtet (wodurch weitere Kopien von dem `auto_ptr` gemacht werden, der als Pivot-Element ausgewählt wurde und gar nichts mehr besitzt), und wenn die Sortierung abgeschlossen ist, wird das Pivot-Element zerstört und Sie haben ein Problem. Wenigstens ein `auto_ptr` in Ihrer Sequenz besitzt nicht mehr länger den Zeiger, den er ursprünglich gehalten hatte, und das zugehörige Objekt ist bereits gelöscht worden.

So hat das Standard-Komitee große Anstrengungen unternommen alles zu tun, um Ihnen hier zu helfen. Der Standard-`auto_ptr` wurde absichtlich und spezifisch so entworfen, dass er nicht funktioniert, wenn Sie versuchen ihn mit einem Standard-Container zu benutzen (oder zumindest bei den meisten natürlichen Implementierungen der Standardbibliothek). Um dies zu erreichen, hat sich das Komitee eines Tricks bedient: `auto_ptr`'s Copy-Konstruktor und Copy-Zuweisungsoperator erwarten eine Referenz auf ein nicht-konstantes Objekt auf der rechten Seite. Die meisten Implementierungen der Standard-Container `insert()`-Funktion für ein einzelnes Element erwarten ein `const` und funtkionieren deshalb nicht mit `auto_ptr`.

Die Sache mit besitzlosen auto_ptr-Objekten

```
    // nachdem a in einen anderen auto_ptr kopiert wurde
    cout << a->Value();
}
```

(Wir nehmen an, dass a kopiert wurde, aber der Zeiger noch nicht von `vector` oder der `sort`-Funktion gelöscht wurde.) Das Kopieren eines `auto_ptr` überträgt nicht nur den Besitz, sondern setzt auch den Quell-`auto_ptr` auf Null zurück. Dies wurde absichtlich so gemacht, um zu verhindern, dass irgendjemand mit einem besitzlosen `auto_ptr` irgendetwas anstellen kann. Einen besitzlosen `auto_ptr` so wie oben zu benutzen ist nicht erlaubt und der Versuch, einen solchen Null-Zeiger zu dereferenzieren, führt zu undefiniertem Verhalten (auf den meisten Systemen zu einem Core-Dump oder einer Speicherzugriffsverletzung).

Dies bringt uns zu der letzten gebräuchlichen Verwendung von `auto_ptr`.

Einpacken von Zeiger-Elementvariablen

```
class C
{
public:    /*...*/
protected: /*...*/
private:   /*...*/
  auto_ptr<CImpl> pimpl_;
};
```

Möglicher Knackpunkt: In einem der `/*...*/`-Bereiche (egal ob `public`, `protected`, oder `private`) sollten sich besser zumindest die Deklarationen für den Copy-Konstruktor und den Copy-Zuweisungsoperator befinden.

`auto_ptr` ist nützlich, um Zeiger-Elementvariablen zu kapseln. Dies funktioniert in etwa so wie in dem einleitenden Beispiel am Anfang dieser Lektion, nur dass wir uns hier nicht am Ende der Funktion den Ärger mit dem Clean-Up sparen, sondern wir sparen uns den Ärger mit dem Clean-Up im Destruktor von `C`.

Wobei es aber immer eins zu beachten gilt: Genauso als wenn Sie einen einfachen Zeiger anstatt des `auto_ptr` als Elementvariable in Ihrer Klasse haben, sollten Sie einen eigenen Destruktor, Copy-Konstruktor und Copy-Zuweisungsoperator für diese Klasse zur Verfügung stellen (auch wenn Sie sie unbrauchbar machen, indem Sie sie als private deklarieren und undefiniert lassen), da das Default-Verhalten für diese Operationen falsch ist.

Das const auto_ptr Idiom

Nachdem wir uns nun durch die tiefen Mysterien von `auto_ptr` gekämpft haben, hier eine Technik, die Sie interessant finden dürften. Neben ihren anderen Vorteilen ist eine Konsequenz der Verbesserung von `auto_ptr`, dass `const auto_ptr`-Objekte niemals ihren Besitz verlieren. Das Kopieren eines `const auto_ptr` ist nicht erlaubt und in der Tat ist das Einzige, was Sie mit einem `const auto_ptr` machen können, ihn mit dem `operator*()` oder dem `operator->()` zu dereferenzieren oder `get()` aufzurufen, um den Wert des enthaltenen Zeigers zu ermitteln. Das heißt wir haben ein klares und knappes Idiom, um auszudrücken, dass ein `auto_ptr` niemals seinen Besitz verlieren kann.

```
const auto_ptr<T> pt1( new T );
    // indem wir pt1 als const definieren, garantieren
    // wir, dass pt1 niemals in einen anderen auto_ptr

    // kopiert werden kann, womit auch garantiert ist,
    // dass pt1 niemals besitzlos werden kann
auto_ptr<T> pt2( pt1 ); // illegal
auto_ptr<T> pt3;
pt3 = pt1;              // illegal
pt1.release();          // illegal
pt1.reset( new T );     // illegal
```

Das ist es, was ich `const` nenne. Wenn Sie also der Welt mitteilen wollen, dass ein `auto_ptr` niemals geändert werden kann und immer das löschen wird, was er besitzt, dies ist der Weg dafür. Das `const-auto_ptr`-Idiom ist eine nützliche und gebräuchliche Technik, die Sie in Erinnerung behalten sollten. Die ursprüngliche Lösung dieser *Guru-of-the-Week*-Ausgabe schloss mit den folgenden Worten: »Dieses `const-auto_ptr`-Idiom ist eines der Dinge, die wahrscheinlich eine verbreitete Technik werden, und Sie können jetzt sagen, dass Sie sie von Anfang an kannten.«

8 Fallen, Stolpersteine und Anti-Idiome

Lektion 38: Objektidentität — Schwierigkeitsgrad: 5

»Wer bin ich wirklich?« Dieses Problem handelt davon, wie Sie entscheiden, ob zwei Zeiger wirklich auf das gleiche Objekt verweisen.

Der »`this != &other`«-Test (wie unten gezeigt) ist gebräuchliche Programmierpraxis, um einer Selbstzuweisung vorzubeugen. Ist die Bedingung nötig und/oder ausreichend, um dies zu erreichen? Warum oder warum nicht? Wenn nicht, wie würden Sie das Problem lösen?

```
T& T::operator=( const T& other )
{
  if( this != &other )  // der fragwürdige Test
  {
    // ...
  }
  return *this;
}
```

Sie sollten unterscheiden zwischen »Sich vor Murphy schützen versus sich vor Machiavelli schützen« – das heißt, sich vor Sachen zu schützen, die von allein schief gehen könnten, versus sich vor einem boshaften Programmierer zu schützen, der unbedingt versucht Ihren Code lahm zu legen.

Lösung

Die kurze Antwort: In den meisten Fällen ist es nur schlimm, wenn Sie den Test nicht durchführen und Selbstzuweisung nicht richtig behandeln. Obwohl dieser Test nicht gegen alle möglichen Fälle schützt, funktioniert er in der Praxis gut genug, solange er nur zu Optimierungszwecken eingesetzt wird. In der Vergangenheit wurde erzählt, das Mehrfachvererbung die Richtigkeit dieses Codes beeinflusst. Das ist nicht wahr, sondern eine Ente.

Exception-Sicherheit (Murphy)

Wenn `T::operator=()` exception-sicher ist, wird der Test auf Selbstzuweisung nicht benötigt. Punkt. Da wir immer exception-sicheren Code schreiben sollten, sollten wir den Selbstzuweisungs-Test nicht mehr durchführen, oder?

☑ Richtlinie

Schreiben Sie niemals einen Copy-Zuweisungsoperator, der sich auf den Test auf Selbstzuweisung verlässt, damit er richtig funktioniert; ein exception-sicherer Copy-Zuweisungsoperator ist automatisch sicher vor Selbstzuweisung.

Ja und nein. Es stellt sich heraus, dass es zwei potenzielle Effizienz-Nachteile gibt, wenn Sie nicht auf Selbstzuweisung prüfen.

- Wenn Sie auf Selbstzuweisung testen können, können Sie die Zuweisung komplett wegoptimieren.
- Oftmals führt Code exception-sicher zu machen auch dazu, ihn weniger effizient zu machen (das »Paranoia-hat-ihren-Preis-Prinzip«).

In der Praxis, wenn Selbstzuweisung oft auftritt (was in den meisten Programmen selten der Fall ist) und der Geschwindigkeitsvorteil, der durch das Unterdrücken der unnötigen Arbeit bei einer Selbstzuweisung erzielt wird, für Ihre Anwendung von Signifikanz ist (noch seltener für die meisten Programme), so testen Sie auf Selbstzuweisung.

☑ Richtlinie

Es ist in Ordnung, auf Selbstzuweisung zu testen, als Optimierung, um unnötige Arbeit zu vermeiden.

Überladen von Operatoren (Machiavelli?)

Da Klassen ihren eigenen `operator&()` zur Verfügung stellen können, könnte der zur Frage gestellte Test etwas völlig anderes bewirken, als beabsichtigt ist. Das fällt unter die Überschrift »Schützen vor Machiavelli«, obwohl man davon ausgehen sollte, dass der Autor von `T::operator=()` weiß, ob seine Klasse auch `T::operator&()` überlädt. Natürlich könnte auch eine Basisklasse einen `operator&()` haben, der den Test kaputtmacht.

Beachten Sie, dass, während ein Klasse auch einen `operator!=()` definieren kann, dies irrelevant ist, da dies nicht mit dem »`this != &other`«-Test interferieren kann. Der Grund hierfür ist, dass Sie keinen `operator!=()` schreiben können, der zwei `T*`-Parameter erwartet, da wenigstens ein Parameter eines überladenen Operators vom Typ der Klasse sein muss.

8.0.1 Postscript #1:

Als Bonus hier ein kleiner Code-Witz. Ob Sie es glauben oder nicht, dies wurde von gut meinenden (aber fehlgeleiteten) Programmierern versucht.

```
T::T( const T& other )
{
  if( this != &other )
  {
    // ...
  }
}
```

Haben Sie es beim ersten Mal Lesen bemerkt?[1]

8.0.2 Postscript #2:

Beachten Sie, dass es andere Fälle gibt, in denen der Vergleich von Zeigern nicht das ist, was die meisten Leute intuitiv finden würden. Hier ein paar Beispiele:

- Der Vergleich von Zeigern auf String-Literale ist undefiniert. Der Grund hierfür ist, dass der Standard Compilern explizit erlaubt, String-Literale als Speicherplatzoptimierung in überlappenden Speicherbereichen abzulegen. Aus diesem Grund ist es absolut möglich, zwei Zeiger zu haben, die in zwei verschiedene String-Literale verweisen und bei einem Vergleich trotzdem gleich sind.
- Im Allgemeinen können Sie nicht beliebige Zeiger nehmen und sie mit den eingebauten Operatoren <, <=, > und >= vergleichen und definierte Resultate erwarten, auch wenn die Resultate in spezifischen Situationen definiert sind (zum Beispiel

1. Das Überprüfen auf Selbstzuweisung macht keinen Sinn in einem Konstruktor, da das `other`-Objekt nicht das gleiche sein kann wie unser eigenes Objekt – unser eigenes Objekt wird ja gerade erst konstruiert! Als ich diesen Code-Witz das erste Mal aufgeschrieben habe, dachte ich (richtigerweise), dass der Test bedeutungslos wäre. Dennoch stellte sich heraus, dass er in die Kategorie »Schützen vor Machiavelli« fällt. Mein Freund und scharfsinniger Leser Jim Hyslop wies darauf hin, dass es ein Stück (technisch nicht erlaubten) Code gibt, der ein Placement-`new` benutzt und, wenn er legal wäre, den Test bedeutungsvoll werden lassen würde:
   ```
   T t;
   new (&t) T(t); // illegal, würde den Test aber bedeutungsvoll machen
   ```
   Ups! Ich bin mir nicht sicher, ob ich glücklich sein soll, dass Leute meinen Code-Witz zu schätzen wissen, oder tief verunsichert, dass sie sofort versucht haben subversive Wege zu finden, um ihm Bedeutung zukommen zu lassen.

können Zeiger auf Objekte im gleichen Array so verglichen werden). Die Standard-Bibliothek umgeht diese Limitierung, indem sie sagt, dass das Vergleichsfunktions-Template der Bibliothek, `less<>` und Verwandschaft, eine definitive Reihenfolge für Zeiger zur Verfügung stellen muss. Dies ist notwendig, so dass Sie, sagen wir, eine `map` mit Schlüsseln, die Zeiger sind, erzeugen können – zum Beispiel `map<T*,U>`, was mit dem Default-Templateparameter das Gleiche ist wie `map<T*,U,less<T*> >`.

Lektion 39: Automatische Umwandlungen — Schwierigkeitsgrad: 4

Automatische Umwandlungen von einem Typ in einen anderen können sehr bequem sein. Diese Lektion behandelt ein typisches Beispiel, um zu illustrieren, warum sie so gefährlich sein können.

`string` aus dem C++-Standard hat keinen impliziten Umwandlungsoperator in einen `const char*`. Sollte es ihn geben?

Es ist oft nützlich, auf einen `string` wie auf einen `const char*` im C-Stil zugreifen zu können. In der Tat verfügt `string` über die Elementfunktion `c_str()`, um Ihnen dies zu erlauben, indem sie Ihnen Zugriff auf ein `const char*` bietet. Hier der Unterschied im Client-Code:

```
string s1("hello"), s2("world");
strcmp( s1, s2 );               // 1 (falsch)
strcmp( s1.c_str(), s2.c_str() ) // 2 (ok)
```

Es wäre mit Sicherheit schön, #1 schreiben zu können, aber #1 ist ein Fehler, da `strcmp` zwei Zeiger erwartet und es keine automatische Umwandlung von `string` in `const char*` gibt. Zeile Nummer 2 ist in Ordnung, ist aber länger, da explizit `c_str()` aufgerufen werden muss.

Somit läuft die Frage dieser Lektion darauf hinaus: Wäre es nicht schöner, wenn wir einfach #1 schreiben könnten?

Lösung

Die Frage war: `string` aus dem C++-Standard hat keinen impliziten Umwandlungsoperator in einen `const char*`. Sollte es ihn geben?

Die Antwort ist: Nein, das hat seinen guten Grund.

Es ist fast immer eine gute Idee, keine automatischen Umwandlungen zu erlauben, sei es mittels Umwandlungsoperatoren oder durch einen nicht als `explicit` deklarierten Konstruktor mit nur einem Argument.[2] Hier die Hauptgründe, warum implizite Umwandlungen im Allgemeinen nicht sicher sind:

- Implizite Umwandlungen können mit der Auflösung von überladenen Funktionen interferieren.
- Implizite Umwandlungen können stillschweigend »falschen« Code sauber kompilieren lassen.

Wenn `string` eine automatische Umwandlung in `const char*` bieten würde, könnte diese Umwandlung implizit überall aufgerufen werden, wo der Compiler es für nötig halten würde. Dies bedeutet, dass Sie alle möglichen Arten subtiler Umwandlungsprobleme bekommen würden – die gleichen, die Sie mit nicht-expliziten Umwandlungs-Konstruktoren bekommen. Es passiert einfach zu schnell, dass man Code schreiben kann, der richtig aussieht, es aber in Wirklichkeit nicht ist und fehlschlagen sollte, aber durch schieren Zufall trotzdem kompiliert wird und dabei etwas völlig anderes macht, als beabsichtigt war.

Dafür gibt es viele gute Beispiele, hier ein einfaches:

```
string s1, s2, s3;
s1 = s2 - s3;   // ups, sollte ein "+" sein
```

Die Subtraktion hat keine Bedeutung und sollte fehlschlagen. Wenn `string` jedoch eine implizite Umwandlung in `const char*` ermöglichen würde, würde dieser Code sauber kompiliert werden, da der Compiler stillschweigend bei Strings in `const char*`-Zeiger umwandeln und diese dann subtrahieren würde.

☑ Richtlinie

Vermeiden Sie es, Umwandlungsoperatoren zu schreiben. Vermeiden Sie nicht-explizite Konstruktoren.

Lektion 40: Objekt-Lebenszeiten – Teil 1 Schwierigkeitsgrad: 5

»Sein oder nicht sein ...« Wann existiert ein Objekt wirklich? In diesem Problem geht es darum, wann es sicher ist, ein Objekt zu benutzen.

Kritisieren Sie das folgende Code-Fragment:

2. Diese Lösung konzentriert sich auf das allgemeine Problem impliziter Typumwandlungen, es gibt jedoch noch andere Gründe, warum die `string`-Klasse keinen `const char*`-Umwandlungsoperator haben sollte. Hier einige Werke, die weiterführende Informationen enthalten: Koenig A. und Moo B. »Ruminations on C++« (Addison-Wesley Longman, 1997), Seiten 290-292. Stroustrup B. »The Design and Evolution of C++« (Addison-Wesley 1994), Seite 83.

```
void f()
{
  T t(1);
  T& rt = t;
  //--- #1: mache etwas mit t oder rt ---
  t.~T();
  new (&t) T(2);
  //--- #2: mache etwas mit t oder rt ---
} // t wird wieder zerstört
```

Ist der Code in Block #2 sicher und/oder legal? Erklären Sie.

Lösung

Ja, der Code in Block #2 ist sicher und legal (wenn Sie bis dahin gelangen), aber:

- Die Funktion als Ganzes ist nicht sicher.
- Es ist eine schlechte Angewohnheit, in die man nicht hineingeraten sollte.

Der C++-Standard erlaubt explizit solchen Code. Die Referenz `rt` wird durch die explizite Destruktion und Rekonstruktion nicht ungültig. (Natürlich können Sie `t` oder `rt` nicht zwischen den Aufrufen von `t.~T()` und dem Placement-`new` benutzen, weil während dieser Zeitspanne kein Objekt existiert. Wir sind auch dazu gezwungen, davon auszugehen, dass `T::operator&()` nicht überladen wurde und etwas anderes macht, als die Adresse des Objekts zurückzugeben.)

Der Grund, warum ich gesagt habe »wenn Sie bis dahin gelangen«, ist, dass `f()` als Ganzes nicht exception-sicher sein könnte.

Die Funktion ist nicht sicher. Wenn `T`'s Konstruktor während der Ausführung von `new (&t) T(2)` eine Exception werfen könnte, dann ist `f()` nicht exception-sicher. Betrachten wir warum: Wenn der Aufruf von `T(2)` zum Werfen einer Exception führt, wurde noch kein neues Objekt in dem Speicherbereich von `t` rekonstruiert, trotzdem wird am Ende der Funktion natürlich `T::~T()` aufgerufen (da `t` ja eine automatische Variable ist), womit »`t` wieder zerstört wird«, wie es der Kommentar besagt. Das heißt, t wird einmal konstruiert, aber zweimal zerstört (ups). Dies führt sehr wahrscheinlich zu unvorhersagbaren Seiteneffekten wie zum Beispiel Core-Dumps.

☑ Richtlinie

Bemühen Sie sich immer exception-sicheren Code zu schreiben. Strukturieren Sie Ihren Code immer so, dass Ressourcen korrekt wieder freigegeben werden und Daten sich in einem konsistenten Zustand befinden, selbst wenn eine Exception auftritt.

Das ist eine schlechte Angewohnheit. Wenn von dem Aspekt der Exception-Sicherheit abgesehen wird, funktioniert der Code in diesem Fall, da der Programmierer den richtigen Typ des Objekts kennt, das zerstört und wieder rekonstruiert wird. Das heißt, das Objekt `t` war ein Objekt vom Typ `T` und wird auch als ein `T` zerstört und wieder rekonstruiert.

Diese Technik ist selten, fall überhaupt jemals, in echtem Code wirklich notwendig und eine schlechte Angewohnheit, in die man nicht reingeraten sollte, da sie voller (manchmal subtiler) Gefahren ist, wenn sie in einer Elementfunktion angewandt wird.

```
// Können Sie das subtile Problem ausmachen?
//
void T::DestroyAndReconstruct( int i )
{
  this->~T();
  new (this) T(i);
}
```

Ist das eine sichere Technik? Im Allgemeinen nicht. Betrachten Sie den folgenden Code.

```
class U : public T { /* ... */ };
void f()
{
  /*AAA*/ t(1);
  /*BBB*/& rt = t;
  //--- #1: mache etwas mit t oder rt ---
  t.DestroyAndReconstruct(2);
  //--- #2: mache etwas mit t oder rt ---
} // t wird wieder zerstört
```

Wenn »`/*AAA*/`« »`T`« ist, funktioniert der Code in #2 immer noch, selbst wenn »`/*BBB*/`« keine »`T`« ist (es könnte eine Basisklasse von T sein).

Wenn »`/*AAA*/`« jedoch »`U`« ist, ist alles aus, egal was »`/*BBB*/`« ist. Das Beste, worauf Sie hoffen können, ist ein sofortiger Core-Dump, da der Aufruf von `t.f()` das Objekt »zerschneidet«. Hierbei meint zerschneiden, dass `t.f()` das ursprüngliche Objekt mit einem neuen Objekt eines anderen Typs ersetzt – in diesem Fall ein `T` statt eines `U`. Selbst wenn Sie gewillt sind, nicht-portablen Code zu schreiben, gibt es keine Möglichkeit zu wissen, ob das Objekt-Layout eines `T`-Objekts überhaupt als `U`-Objekt benutzbar ist, wenn es im Speicher darübergelegt wird, wo vorher ein `U`-Objekt war. Die Chancen stehen gut dafür, dass dem nicht so ist.

Machen Sie so etwas nicht, es ist keine gute Programmierpraxis.

☑ Richtlinie

Vermeiden Sie die »dunklen Ecken« einer Programmiersprache; benutzen Sie die einfachsten Techniken, die effektiv sind.

Diese Lektion hat einige fundamentale Sicherheitsprobleme von In-Place-Destruktion und Rekonstruktion aufgedeckt. Damit ist der Grundstein für die folgende Frage in Lektion 41 gelegt.

Lektion 41: Objekt-Lebenszeiten – Teil 1 Schwierigkeitsgrad: 6

Als Fortsetzung von Lektion 40 wird hier ein C++-Idiom betrachtet, das häufig empfohlen wird, das aber oft gefährlich falsch ist.

Kritisieren Sie das folgende »Anti-Idiom« (hier wird gezeigt, wie es oft präsentiert wird).

```
T& T::operator=( const T& other )
{
  if( this != &other )
  {
    this->~T();
    new (this) T(other);
  }
  return *this;
}
```

1. Welches legitime Ziel wird zu erreichen versucht? Korrigieren Sie alle Code-Mängel dieser Version.
2. Ist dieses Idiom selbst nach Korrektur aller Code-Mängel sicher? Erklären Sie. Falls nicht, wie sollte ein Programmierer dann das beabsichtigte Resultat erreichen?

   (Siehe auch Lektion 40.)

Lösung

Dieses Idiom wird häufig empfohlen und erscheint als Beispiel im C++-Standard (siehe die Diskussion im folgenden Kasten). Es ist aber auch sehr armselig und verursacht einen Haufen Probleme. Machen Sie so etwas nicht.

Zum Glück gibt es, wie wir sehen werden, einen richtigen Weg, um den gewünschten Effekt zu erreichen.

Ein warnendes Beispiel

Im C++-Standard ist die Absicht dieses Beispiels, die Lebenszeit-Regeln für ein Objekt zu demonstrieren, nicht jedoch eine gute Programmierpraxis zu empfehlen (es ist keine). Für die Interessenten hier eine etwas gekürzte Version aus Abschnitt 3.8, Absatz 7:

```
[Example:
  struct C {
    int i;
    void f();
    const C& operator=( const C& );
  };
  const C& C::operator=( const C& other)
  {
    if ( this != &other )
    {
      this->~C();     // lifetime of *this ends
      new (this) C(other);
                      // new object of type C created
      f();            // well-defined
    }
    return *this;
  }
  C c1;
  C c2;
  c1 = c2; // well-defined
  c1.f();  // well-defined; c1 refers to a new object of type C
--end example]
```

Als weiterer Beweis dafür, dass dies nicht als Beispiel für gute Programmierpraxis gedacht ist, beachten Sie, dass `C::operator=()` ein `const C&` statt eines einfachen `C&` zurückgibt, was unnötigerweise die portable Benutzung dieser Objekte in den Containern der Standardbibliothek unterbindet.

Aus den PeerDirect Coding-Standards:

- Deklarieren Sie die Copy-Zuweisung als `T& T::operator=(const T&)`
- Geben Sie kein `const T&` zurück. Obwohl dies schön wäre, da es einem Missbrauch wie in »`(a=b)=c`« vorbeugt, würde es auch bedeuten, dass Sie zum Beispiel `T`-Objekte nicht portabel in Container der Standardbibliothek packen könnten, da diese es erfordern, dass der Zuweisungsoperator ein einfaches `T&` zurückliefert. (siehe auch Cline95: 212; Murray93: 32-33)

Lassen Sie uns jetzt noch mal die Fragen betrachten.

1. Welches legitime Ziel wird zu erreichen versucht?

   Dieses Idiom drückt die Copy-Zuweisung mit Hilfe des Copy-Konstruktors aus. Das heißt, es wird versucht sicherzustellen, dass `T`'s Copy-Zuweisung das Gleiche macht wie `T`'s Copy-Konstruktor, wodurch der Programmierer nicht den gleichen Code doppelt in zwei verschiedenen Plätzen schreiben muss.

   Dies ist ein nobles Ziel. Schließlich macht es das Programmieren leichter, wenn Sie den gleichen Code nicht doppelt schreiben müssen, und wenn sich `T` ändert (zum Beispiel wenn `T` eine neue Elementvariable hinzugefügt wird), kann es Ihnen nicht mehr passieren, dass Sie eine der Funktionen aktualisieren und die andere vergessen.

   Dieses Idiom könnte insbesondere dann nützlich sein, wenn es virtuelle Basisklassen gibt, die Elementvariablen haben, die ansonsten falsche Zuweisungen oder im besten Fall mehrfache Zuweisungen bekommen würden. Auch wenn sich das gut anhört, ist es kein wirklicher Vorteil in der Realität, da virtuelle Basisklassen sowieso keine Elementvariablen besitzen sollten (siehe Meyers98, Meyers99 und Barton94). Außerdem, wenn es virtuelle Basisklassen gibt, bedeutet dies, dass die Klasse zur Vererbung entworfen wurde, was (wie wir gleich sehen werden) bedeutet, dass wir das Idiom nicht benutzen können – es ist zu gefährlich.

   Der letzte Teil der ersten Frage war: Korrigieren Sie alle Code-Mängel dieser Version. Der obige Code hat einen Mangel, der korrigiert werden kann, und einige andere, die nicht korrigierbar sind.

Problem 1: Objekte können zerschnitten werden

Der Code »`this->~T(); new (this) T(other);`« macht nicht das Richtige, wenn `T` eine Basisklasse mit einem virtuellen Destruktor ist. Wenn diese Zeile für ein Objekt einer abgeleiteten Klasse aufgerufen wird, wird das Objekt der abgeleiteten Klasse zerstört und mit einem `T`-Objekt ersetzt. Dies lässt mit großer Sicherheit folgenden Code fehlschlagen, der versucht, dieses Objekt zu benutzen. Siehe Lektion 40 für weitergehende Informationen zu diesem Problem.

Insbesondere wird hierdurch das Leben von Autoren abgeleiteter Klassen unglaublich schwer gemacht (und es gibt andere potenzielle Fallen bei abgeleiteten Klassen, siehe weiter unten). Erinnern Sie sich, dass abgeleitete Zuweisungsoperatoren normalerweise mit Hilfe des geerbten Zuweisungsoperators wie folgt implementiert werden:

```
Derived&
Derived::operator=( const Derived& other )
{
  Base::operator=( other );
```

```
  // ...hier werden die neuen Elementvariablen
  //    von Derived zugewiesen...
  return *this;
}
```

In diesem Fall erhalten wir:

```
class U : T { /* ... */ };
U& U::operator=( const U& other )
{
  T::operator=( other );
  // ...hier werden die neuen Elementvariablen
  //    von U zugewiesen...
  // ...ups, aber wir haben gar kein U-Objekt mehr!
  return *this;  // ebenfalls ups!
}
```

So, wie anfangs beschrieben, würde der Aufruf von `T::operator=()` stillschweigend allen folgenden Code fehlschlagen lassen (sowohl die Zuweisungen der neuen Elementvariablen von `U` als auch die `return`-Anweisung). Dies manifestiert sich oft in einem mysteriösen und schwer zu findenden Laufzeitfehler, wenn der Destruktor von `U` die Elementvariablen nicht auf ungültige Werte zurücksetzt.

Um das Problem zu korrigieren, könnten wir einige Alternativen ausprobieren.

- Sollte stattdessen `U`'s Copy-Zuweisungsoperator »`this->T::~T();`« aufrufen, vielleicht gefolgt von einem Placement-`new` für das `T`-Basisunterobjekt? Nun, hiermit würden wir sicherstellen, dass für ein abgeleitetes Objekt nur das `T`-Basisunterobjekt ersetzt werden würde (anstatt dass das ganze abgeleitete Objekt zerschnitten werden würde und fälschlicherweise in ein `T`-Objekt überführt werden würde). Es stellt sich heraus, dass dieser Versuch selbst einige Falltüren hat, lassen Sie uns aber die Idee erst mal festhalten.

- Sollten wir also einen Schritt weiter gehen und in `U`'s Copy-Zuweisungsoperator `T`'s Idiom der In-Place-Destruktion (des ganzen U-Objekts) und Rekonstruktion folgen? Dies ist eine bessere Alternative als die vorhergehende, illustriert aber sehr gut eine weitere Schwäche des Idioms. Wenn eine Basisklasse das Idiom benutzt, müssen auch alle abgeleiteten Klassen das Idiom benutzen. (Dies ist aus vielen Gründen keine gute Sache, wovon einer der ist, dass der vom Compiler generierte Default-Zuweisungsoperator dieses Idiom nicht benutzt und viele Programmierer, die eine Klasse ableiten, ohne Elementvariablen hinzuzufügen, gar nicht erst daran denken, eine eigene Version des `operator=()` zur Verfügung zu stellen, da sie davon ausgehen, dass die vom Compiler generierte Version in Ordnung sein sollte.)

Hmm, beide dieser möglichen Alternativen ersetzen die eine offensichtliche Gefahr nur durch eine subtilere Gefahr, die immer noch Autoren abgeleiteter Klassen treffen kann (siehe unten).

☑ Richtlinie

Vermeiden Sie die »dunklen Ecken« einer Programmiersprache; benutzen Sie die einfachsten Techniken, die effektiv sind.

☑ Richtlinie

Vermeiden Sie unnötig knappen oder cleveren Code, selbst wenn er Ihnen völlig klar ist, wenn Sie ihn zum ersten Mal schreiben.

2. Nun zur zweiten Frage: Ist dieses Idiom selbst nach Korrektur aller Code-Mängel sicher? Erklären Sie.

   Nein. Beachten Sie, dass keines der folgenden Probleme gelöst werden kann, ohne das Idiom völlig aufzugeben.

Problem 2: Es ist nicht exception-sicher

Die `new`-Anweisung ruft den Copy-Konstruktor von `T` auf. Falls dieser Konstruktor eine Exception werfen könnte (und viele Klassen zeigen einen Fehler während der Konstruktion durch das Werfen einer Exception an), dann ist die Funktion nicht exception-sicher, da sie das alte Objekt zerstört, ohne es durch irgendwas zu ersetzen.

Wie beim Zerschneiden des Objekts, so lässt auch dieser Mangel folgenden Code, der versucht das Objekt zu benutzen, fehlschlagen. Noch schlimmer, wahrscheinlich wird das Programm versuchen das gleiche Objekt zweimal zu zerstören, da der Code außerhalb der Funktion nicht wissen kann, dass der Destruktor für das Objekt schon aufgerufen wurde. (Siehe Lektion 40 für weitergehende Information zur doppelten Zerstörung.)

☑ Richtlinie

Bemühen Sie sich immer exception-sicheren Code zu schreiben. Strukturieren Sie Ihren Code immer so, dass Ressourcen korrekt wieder freigegeben werden und Daten sich in einem konsistenten Zustand befinden, selbst wenn eine Exception auftritt.

Problem 3: Es ändert die normale Lebenszeit des Objekts

Dieses Idiom macht allen Code untauglich, der sich auf die normale Lebenszeit des Objekts verlässt. Insbesondere zerstört oder interferiert es mit allen Klassen, welche

das verbreitete »Ressourcenerwerb ist Initialisierung«-Idiom verwenden. Generell steht es im Widerspruch zu allen Klassen, deren Konstruktor oder Destruktor Seiteneffekte haben.

Was ist zum Beispiel, wenn `T` (oder irgendeine Basisklasse von `T`) eine Mutex-Sperre erwirbt oder eine Datenbank-Transaktion im Konstruktor startet und im Destruktor die Sperre freigibt oder die Transaktion beendet? Dann wird die Sperre/Transaktion fälschlicherweise beendet und während der Zuweisung neu angefordert/gestartet – was typischerweise sowohl den Client-Code als auch die Klasse selbst unbrauchbar macht. Neben `T` und `T`'s Basisklasse(n) kann dies auch alle abgeleiteten Klassen von `T` betreffen, wenn sie auf die normale Lebenszeit von `T`-Objekten vertrauen.

Einige werden jetzt sagen: »Aber ich würde so etwas nie in einer Klasse machen, die in ihrem Konstruktor/Destruktor eine Sperre erwirbt/freigibt!« Die Antwort darauf lautet: »Wirklich? Und woher wissen Sie, dass keine Ihrer direkten oder indirekten Basisklassen so etwas tut?« Kurz gesagt können Sie dies meistens nicht wissen und Sie sollten sich nie darauf verlassen, dass Ihre Basisklassen immer noch richtig funktionieren, wenn Sie mit den Lebenszeiten Ihrer Objekte spielen.

Das grundlegende Problem ist, dass dieses Idiom die Bedeutung von Konstruktion und Destruktion unterminiert. Konstruktion und Destruktion korrespondieren exakt mit dem Anfang und Ende der Lebenszeit eines Objekts, zu deren Zeitpunkt typischerweise Ressourcen erworben und freigegeben werden. Konstruktion und Destruktion sind nicht dazu gedacht, den Wert eines Objektes zu verändern (was sie ja auch gar nicht machen; eigentlich zerstören sie das alte Objekt und ersetzen es durch ein ähnlich aussehendes, das den neuen Wert besitzt, was ganz und gar nicht das Gleiche ist).

Problem 4: Es kann immer noch abgeleitete Klassen untauglich machen

Wenn Problem 1 mit dem ersten Ansatz gelöst wird, nämlich indem direkt »`this->T::~T();`« aufgerufen wird, ersetzt das Idiom nur den »`T`-Teil« (oder `T`-Basisunterobjekt) innerhalb des abgeleiteten Objekts. Zum einen sollte uns das nervös machen, da es die normale Garantie von C++ unterminiert, dass die Lebenszeit von Basisunterobjekten die Lebenszeit aller abgeleiteten Objekte vollständig umfasst – dass also Basisunterobjekte immer vor dem kompletten abgeleiteten Objekt konstruiert und immer erst danach zerstört werden. In der Praxis werden viele abgeleitete Klassen damit klarkommen, dass die von ihren Basisklassen geerbten Teile ausgetauscht werden, aber bei einigen Klassen könnte dies nicht der Fall sein.

Insbesondere könnten alle abgeleiteten Klassen unbrauchbar werden, die die Verantwortung für den Status ihrer Basisklasse übernehmen, wenn der von der Basisklasse

geerbte Teil ohne Wissen der abgeleiteten Klasse geändert wird (und stillschweigend ein Objekt zu zerstören und durch ein neu konstruiertes zu ersetzen zählt bestimmt als eine solche Änderung). Diese Gefahr kann verringert werden, solange der Zuweisungsoperator nichts Ungewöhnliches oder Unerwartetes macht, was ein »normal geschriebener« Zuweisungsoperator nicht machen würde.

Problem 5: this != &other

Dieses Idiom verlässt sich vollständig auf den »`this != &other`«-Test. (Wenn Sie das bezweifeln, überlegen Sie sich, was im Falle einer Selbstzuweisung passieren würde, gäbe es diesen Test nicht.)

Das Problem ist das gleiche, welches schon in Lektion 38 behandelt wurde: Jeder Copy-Zuweisungsoperator, der auf Selbstzuweisung prüfen *muss*, ist nicht exceptionsicher.[3]

Dieses Idiom leidet an anderen potenziellen Gefahren, die Client-Code und/oder abgeleitete Klassen beeinflussen können (wie das Verhalten, wenn es virtuelle Zuweisungsoperatoren gibt, die sowieso meistens ziemlich verzwickt sind), aber diese Probleme sollten gereicht haben, Ihnen zu demonstrieren, dass das Idiom mit einigen ernsten Problemen zu kämpfen hat.

Und nun zum letzten Teil der zweiten Frage: Falls nicht, wie sollte ein Programmierer dann das beabsichtige Resultat erreichen?

Den Copy-Zuweisungsoperator mit Hilfe des Copy-Konstruktors zu verwirklichen ist eine gute Idee, aber der richtige Weg, dies zu implementieren, besteht darin, die klare (um nicht zu sagen elegante) Form des stark exception-sicheren Copy-Zuweisungsoperators zu benutzen. Stellen Sie eine `Swap()`-Elementfunktion zur Verfügung, die garantiert keine Exception wirft und einfach nur als atomare Operation die Innereien zweier Objekte miteinander vertauscht. Danach schreiben Sie den Copy-Zuweisungsoperator wie folgt:

```
T& T::operator=( const T& other )
{
  T temp( other );  // alle Arbeit im Hintergrund erledigen,
  Swap( temp );     // dann das Arbeitsergebnis nur mit
  return *this;     // Operationen offen legen, die keine
}                   // Exception werfen können
```

3. Es ist nichts falsch daran, den »`this != &other`«-Test als Optimierung im Falle einer Selbstzuweisung zu benutzen. Wenn er funktioniert, haben Sie sich eine Zuweisung erspart. Wenn nicht, sollte Ihr Zuweisungsoperator natürlich dennoch so geschrieben sein, dass er auch bei einer Selbstzuweisung sicher funktioniert. Es gibt sowohl Argumente für als auch gegen die Verwendung dieses Tests als Optimierung, was aber über den Rahmen dieser Lektion hinausgehen würde.

Diese Methode implementiert den Copy-Zuweisungsoperator immer noch mit Hilfe des Copy-Konstruktors, macht dies aber richtig, sicher, leicht und gesund. Es werden keine Objekte zerschnitten; der Zuweisungsoperator ist stark exception-sicher, es wird nicht mit den Lebenszeiten von Objekten gespielt und man ist nicht vom Test auf Selbstzuweisung abhängig.

Für weitere Informationen zu dieser klaren Form und den verschiedenen Stufen von Exception-Sicherheit siehe auch Lektionen 8 bis 17 sowie 40.

Somit lässt sich zusammenfassend sagen, dass das ursprüngliche Idiom voller Fallen war; es ist oft falsch und erschwert das Leben von Autoren abgeleiteter Klassen unglaublich. Ich bin manchmal versucht den Code in der Büroküche aufzuhängen mit der Überschrift: »Vorsicht, Drachen!«[4]

☑ Richtlinie

Bevorzugen Sie es, eine `Swap()`-Elementfunktion zur Verfügung zu stellen, die garantiert keine Exception wirft, und implementieren Sie den Copy-Zuweisungsoperator mit Hilfe des Copy-Konstruktors wie folgt:

```
// GUT
T& T::operator=( const T& other )
{
  T temp( other );
  Swap( temp );
  return *this;
}
```

4. Ja, keine der hier präsentierten Lösungen funktioniert mit Klassen, die Referenzen als Elementvariablen haben – aber das ist keine Limitierung der Lösungen, sondern eine direkte Folge der Tatsache, dass Klassen mit Referenz-Elementvariablen nicht zugewiesen werden sollten. Wenn jemand das Objekt ändern muss, auf das eine Referenz verweist, sollte stattdessen wirklich ein Zeiger benutzt werden.

☑ Richtlinie

Benutzen Sie niemals den Trick, den Copy-Zuweisungsoperator mit Hilfe des Copy-Konstruktors zu implementieren, indem Sie explizit den Destruktor aufrufen gefolgt von einem Placement-`new`, auch wenn dieser Trick alle drei Monate in der Newsgroup auftaucht – das heißt, schreiben Sie niemals:

```
// SCHLECHT
T& T::operator=( const T& other )
{
  if( this != &other )
  {
    this->~T(); // bösartig
    new (this) T( other );
  }
  return *this;
}
```

Einige Ratschläge der Einfachheit zuliebe

Mein bester Ratschlag? Werden Sie nicht fantasievoll. Behandeln Sie neue C++-Merkmale so, wie Sie eine geladene automatische Waffe in einem Raum voller Leute behandeln würden. Benutzen Sie sie niemals, nur weil es gut aussieht. Warten Sie, bis Sie die Konsequenzen verstehen. Versuchen Sie nicht zu schlau zu sein, schreiben Sie, was Sie wissen, und wissen Sie, was Sie schreiben.

Schlauheit tut weh, aus mindestens zwei Gründen. Erstens ist schlauer Code nicht immer portabel. Der clevere Programmiertrick, den Sie auf einem Compiler gekonnt benutzt haben, mag mit einem anderen nicht mehr funktionieren. Schlimmer noch, der zweite Compiler könnte den Code wortlos akzeptieren, aber zu unbeabsichtigten Ergebnissen führen. Als einfaches Beispiel, viele Compiler unterstützen noch keine Default-Templateparameter oder Template-Spezialisierung, so dass Code, der auf diese Merkmale aufbaut, heutzutage als nicht portabel anzusehen ist. Ein etwas subtileres Beispiel: Viele Compiler sind sich nicht darüber einig, wie mit mehreren gleichzeitig aktiven Exceptions umzugehen ist. Und wenn Sie gerade stutzen – weil Sie dachten, dass mehrere gleichzeitig aktive Exceptions nicht erlaubt sind (ein verbreiteter Irrglaube) –, stellen Sie sich vor, wie die armen Compiler-Hersteller darüber grübeln müssen, da in der Tat beim Aufräumen des Stacks beliebig viele Exceptions gleichzeitig aktiv sein können.

Zweitens, schlauer Code verträgt sich nicht mit guter Wartbarkeit. Erinnern Sie sich immer daran, dass jemand den Code, den Sie schreiben, warten muss. Was Sie heute als tollen Programmiertrick betrachten, wird morgen als unverständliches Rattennest von der Person verflucht, die Ihren Code warten muss – und diese Person könnten sogar Sie sein. Zum Beispiel hat eine Grafikbibliothek einmal ganz »schlau« Gebrauch von dem damals neuen Merkmal, Operatoren zu überladen, gemacht. Wurde diese Bibliothek benutzt, so sah der Client-Code zum Hinzufügen eines Control-Widgets zu einem existierenden Fenster ungefähr so aus:

```
Window  w( /*...*/ );
Control c( /*...*( );
W + c;
```

Anstatt schlau zu sein, hätten sich die Autoren der Bibliothek an Scott Meyers' exzellenten Ratschlag halten sollen »do as the `int`s do«. Das heißt, wenn Sie Operatoren überladen oder ein anderes Sprachmerkmal für Ihre eigenen Klasse benutzen, sollten Sie im Zweifelsfalle Ihre Klasse so schreiben, dass sie der gleichen Semantik wie die eingebauten und Standardbibliotheks-Typen folgt.

Schreiben Sie, was Sie wissen, wenn Sie Code schreiben. Versuchen Sie bewusst 90% des Codes, den Sie schreiben, auf die Sprachmerkmale zu beschränken, die Sie sehr gut verstehen und im Schlaf richtig schreiben könnten, (Sie träumen doch von C++-Code, oder?), und benutzen Sie die anderen 10% um Erfahrung mit anderen Merkmalen zu sammeln. Für diejenigen, für die C++ noch etwas neuer ist, bedeutet dieser Ratschlag zunächst, C++ als besseres C einzusetzen, und daran ist absolut nichts falsch.

Als Nächstes: Wissen Sie, was Sie schreiben. Seien Sie sich der Konsequenzen bewusst, was Ihr Code eigentlich tut. Neben anderen Dingen, lesen Sie weiterhin solche Bücher wie dieses; relevante Internet-Newsgroups wie *comp.lang.c++.moderated* und *comp.std.c++*; Computermagazine wie den *»C++ Report«*, das *»C/C++ User's Journal«* und *»Dr. Dobb's Journal«*. Jeder benötigt diese ständige Weiterbildung um auf dem aktuellen Stand zu bleiben und das Verständnis seiner gewählten Programmiersprache zu vertiefen.

Um die Wichtigkeit zu illustrieren, dass Sie wissen sollen, was Sie schreiben, betrachten Sie das gefährliche Idiom, das wir gerade in dieser Lektion diskutiert haben – nämlich der Trick, den Copy-Zuweisungsoperator mit Hilfe des Copy-Konstruktors und expliziter Destruktion gefolgt von einem Placement-`new` zu implementieren. Dieser Trick taucht regelmäßig alle paar Monate in den C++-Newsgroups auf, und diejenigen, die ihn vertreten, meinen es meistens gut, in der Absicht Code-Duplizierung zu vermeiden – aber wissen diese Programmierer wirklich, was sie schreiben? Wahrscheinlich nicht, da es hier subtile Fallstricke gibt, von denen manche sehr ernst zu nehmen sind.

Diese Lektion hat die meisten aufgedeckt. Es war überhaupt nichts falsch an dem ursprünglichen Ziel, die Konsistenz der Klasse zu verbessern, aber in diesem Fall hätten Sie, wenn Sie gewusst hätten, was Sie schreiben, diese Konsistenz auf einem besseren Weg erreichen können – zum Beispiel mit einer privaten Elementfunktion, die alle Arbeit verrichtet und sowohl vom Copy-Konstruktor als auch vom Copy-Zuweisungsoperator aufgerufen wird.

Kurz gesagt, vermeiden Sie die dunklen Ecken einer Programmiersprache und erinnern Sie sich, dass Schlauheit oft weh tut. Vor allem, schreiben Sie, was Sie wissen, und wissen Sie, was Sie schreiben, und Sie werden wahrscheinlich einen guten Job machen.

9 Verschiedenes

Lektion 42: Initialisierung von Variablen – oder was? Schwierigkeitsgrad: 3

Das erste Problem verdeutlicht die Wichtigkeit zu verstehen, was man schreibt. Hier haben wir vier einfache Zeilen Code, von denen keine zwei das Gleiche bedeuten, obwohl die Syntax nur leicht variiert.

Was ist der Unterschied, falls es einen gibt, zwischen den folgenden Zeilen? (`T` steht dabei für einen Klassentyp.)

```
T t;
T t();
T t(u);
T t = u;
```

Lösung

Diese Aufgabenstellung demonstriert den Unterschied zwischen drei verschiedenen Initialisierungsarten: Default-Initialisierung, direkte Initialisierung und Copy-Initialisierung. Sie enthält auch eine Ausnahme, die gar keine Initialisierung ist. Lassen Sie uns die Fälle einen nach dem anderen betrachten.

```
T t;
```

Hierbei handelt es sich um *Default-Initialisierung*. Dieser Code deklariert eine Variable mit dem Namen und `t` vom Typ `T`, wobei die Variable mit Hilfe des Default-Konstruktors `T::T()` initialisiert wird.

```
T t();
```

Dies ist die Ausnahme. Auf den ersten Blick mag es wie eine weitere Variableninitialisierung aussehen. In Wirklichkeit ist es aber eine Funktionsdeklaration für eine Funktion namens `t`, die keine Argumente hat und ein Objekt vom Typ `T` via Wert zurückgibt. (Wenn Sie dies nicht auf Anhieb sehen, bedenken Sie, dass der obige Code sich nicht von etwas wie `int f();` unterscheidet, was ja wohl offensichtlich eine Funktionsdeklaration ist.)

Einige Leute haben vorgeschlagen »`auto T t();`« zu schreiben, in dem Versuch die `auto`-Speicherklasse zu benutzen, um zu zeigen, dass sie wirklich eine per Default-Konstruktor initialisierte Variable namens `t` vom Typ `T` haben möchten. Erlauben Sie mir einige Bemerkungen hierzu. Zunächst einmal würde dies bei einem standard-konformen Compiler nicht funktionieren; der Compiler würde immer noch eine Funktionsdeklaration erkennen und diese dann abweisen, da die `auto`-Speicherklasse nicht für einen Rückgabewert angegeben werden kann. Als Nächstes, selbst wenn es funktionieren würde, wäre es nicht sinnvoll, da es schon einen einfacheren Weg gibt, um dies auszudrücken: Wenn Sie eine per Default-Konstruktor initialisierte Variable `t` vom Typ `T` wollen, schreiben Sie einfach »`T t;`« und hören Sie auf, die armen Programmierer, die Ihren Code warten müssen, mit solchen Finessen zu verwirren. Ziehen Sie die einfache Lösung immer der schlauen Lösung vor. Schreiben Sie keinen Code, der subtiler als nötig ist.

```
T t(u);
```

Dies ist *direkte Initialisierung*. Die Variable `t` wird direkt mit dem Wert von `u` durch den Aufruf von `T::T(u)` initialisiert.

```
T t = u;
```

Dies ist *Copy-Initialisierung*. Die Variable `t` wird immer mit Hilfe des Copy-Konstruktors von `T` initialisiert, unter Umständen nach dem Aufruf einer anderen Funktion.

☒ Typischer Fehler

Dies ist immer Initialisierung; es ist niemals eine Zuweisung, womit auch niemals T::operator=() aufgerufen wird. Ja, ich weiß, dass dort ein »=« steht, lassen Sie sich aber dadurch nicht hinters Licht führen. Dies ist nur eine von C übernommene Syntax, keine Zuweisungsoperation.

Hier nun die Semantik:

- Wenn `u` vom Typ `T` ist, ist dies das Gleiche, als wenn Sie »`T t(u);`« schreiben würden – es wird einfach nur der Copy-Konstruktor von T aufgerufen.
- Wenn u von einem anderen Typ ist, so hat dies die gleiche Bedeutung, als wenn Sie »`T t( T(u) );`« schreiben würden – das heißt, `u` wird als Erstes in ein temporäres Objekt vom Typ `T` umgewandelt, und danach wird `t` durch den Aufruf des Copy-Konstruktors mit diesem Objekt als Parameter initialisiert. Beachten Sie jedoch, dass es in diesem Fall dem Compiler erlaubt ist, die »Extra«-Kopie wegzuoptimieren und daraus eine Direkt-Initialisierung zu machen (was das Gleiche wäre wie »`T t(u);`«). Wenn Ihr Compiler dies tut, muss dennoch auf den Copy-Konstruktor zugegriffen werden können. Wenn der Copy-Konstruktor jedoch Seiteneffekte hat, könnten es passieren, dass Sie nicht die Ergebnisse bekommen, die Sie erwar-

ten, da die Seiteneffekte des Copy-Konstruktors ausgeführt werden könnten oder auch nicht, je nachdem, ob der Compiler die Optimierung durchführt oder nicht.

☑ Richtlinie

Ziehen Sie die Form »`T t(u);`« der Schreibweise »`T t = u;`« wann immer möglich vor. Die erste funktioniert gewöhnlich überall, wo die zweite geht, und hat noch weitere Vorteile – zum Beispiel können mehrere Parameter übergeben werden.

Lektion 43: `const`-Richtigkeit — Schwierigkeitsgrad: 6

`const` ist ein mächtiges Werkzeug zum Schreiben sichereren Codes. Benutzen Sie `const` so oft wie möglich, aber nicht öfter. Hier einige offensichtliche und nicht so offensichtliche Stellen, wo `const` eingesetzt werden sollte – oder auch nicht.

Kommentieren und ändern Sie die Struktur dieses Programmes nicht; Es wurde nur zu Illustrationszwecken entworfen und komprimiert. Fügen Sie nur (oder entfernen Sie) `const` wo passend (einschließlich kleinerer Änderungen und verwandter Schlüsselwörter).

Bonusfrage: An welcher Stelle sind die Ergebnisse des Programms nicht definiert/nicht kompilierbar durch `const`-Fehler?

```
class Polygon
{
public:
  Polygon() : area_(-1) {}
  void  AddPoint( const Point pt ) { InvalidateArea();
                                     points_.push_back(pt); }
  Point GetPoint( const int i )    { return points_[i]; }
  int   GetNumPoints()             { return points_.size(); }
  double GetArea()
  {
    if( area_ < 0 ) // falls noch nicht berechnet und gecacht
    {
      CalcArea();     // jetzt berechnen
    }
    return area_;
  }
private:
  void InvalidateArea() { area_ = -1; }
  void CalcArea()
  {
    area_ = 0;
    vector<Point>::iterator i;
    for( i = points_.begin(); i != points_.end(); ++i )
      area_ += /* einige Berechnungen */;
```

```
  }
  vector<Point> points_;
  double        area_;
};
Polygon operator+( Polygon& lhs, Polygon& rhs )
{
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
  for( int i = 0; i < last; ++i ) // verknüpfen
  {
    ret.AddPoint( rhs.GetPoint(i) );
  }
  return ret;
}

void f( const Polygon& poly )
{
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}

void g( Polygon& const rPoly ) { rPoly.AddPoint( Point(1,1) ); }

void h( Polygon* const pPoly ) { pPoly->AddPoint( Point(2,2) ); }

int main()
{
  Polygon poly;
  const Polygon cpoly;
  f(poly);
  f(cpoly);
  g(poly);
  h(&poly);
}
```

Lösung

Wenn ich dieses Problem vorstelle, habe ich festgestellt, dass die meisten Leute denken, dass dies ein einfaches Problem ist, und nur die bekannteren `const`-Stolpersteine betrachten. Es gibt jedoch subtilere, die es wert sind, dass man sie kennt, deshalb diese Lektion. Siehe auch den Kasten »Ihre Freunde `const` und `mutable`«.

```
class Polygon
{
public:
  Polygon() : area_(-1) {}
  void  AddPoint( const Point pt ) { InvalidateArea();
                                     points_.push_back(pt); }
```

1. Das `Point`-Objekt wird via Wert übergeben, so dass es wenig bringt, es als `const` zu deklarieren. Für den Compiler ist es sogar egal, ob Sie `const` vor dem Wert-Parameter schreiben, oder nicht – die Signatur der Funktion ist die gleiche. Zum Beispiel:

```
int f( int );
int f( const int );  // Neudeklaration von f(int)
                     // kein Überladen, es gibt nur eine Funktion
int g( int& );
int g( const int& ); // nicht das Gleiche wie g(int&)
                     // g is überladen
```

☑ Richtlinie

Vermeiden Sie, Parameter, die als Wert übergeben werden, als `const` zu deklarieren.

```
Point GetPoint( const int i )    { return points_[i]; }
```

2. Hier gilt genau das Gleiche für den Parameter. Normalerweise ist ein `const` Wert-Parameter bestenfalls sinnlos und irreführend.

3. Dies sollte eine `const`-Elementfunktion sein, da sie den Status des Objekts nicht ändert.

4. Wenn Sie keinen eingebauten Typ wie `int` oder `long` zurückgeben, dann sollten Sie bei einer Rückgabe via Wert immer einen `const`-Wert zurückgeben.[1] Sie unterstützen so Client-Code, indem Sie sicherstellen, dass der Compiler einen Fehler produziert, wenn der Aufrufer versucht, den temporären Wert zu modifizieren.

   Zum Beipiel könnte ein Programmierer so etwas wie »`poly.GetPoint(i) = Point (2,2);`« versuchen. Falls dies funktionieren soll, müsste `GetPoint` eine Referenz und keinen Wert zurückgeben. Wie wir noch sehen werden, macht es für `GetPoint()` aus einem anderen Grund Sinn, einen `const`-Wert oder eine Referenz auf eine `const`-Variable zurückzugeben: GetPoint() sollte für `const Polygon`-Objekte im `operator+()` benutzt werden können.

☑ Richtlinie

Wenn Sie ein Rückgabe via Wert für einen nicht eingebauten Datentypen benutzen, sollten Sie einen `const`-Wert zurückgeben.

```
int   GetNumPoints()             { return points_.size(); }
```

1. Ich möchte darauf hinweisen, dass mir in diesem Punkt nicht jeder zustimmt. In Lakos96 (Seite 618) spricht sich der Autor gegen die Rückgabe eines `const`-Wertes aus und weist darauf hin, dass dies bei den eingebauten Typen (zum Beispiel die Rückgabe von »`const int`«) sowieso redundant ist, und, wie er zu beachten gibt, Schwierigkeiten bei Template-Instanziierungen machen kann.

5. Wiederum sollte diese Funktion `const` sein.

   Zufällig ist dies auch ein gutes Beispiel, wann Sie keinen `const`-Wert zurückgeben sollten. Die Rückgabe eines »`const int`« wäre in diesem Fall falsch, da: (a) die Rückgabe eines »`const int`« Ihnen überhaupt nichts bringt, da der zurückgegebene `int` bereits ein R-Wert ist und somit nicht geändert werden kann; und (b) die Rückgabe von »`const int`« mit der Instanziierung von Templates interferieren könnte und es verwirrend und irreführend ist und wahrscheinlich nur den Code aufbläht.

```
double GetArea()
{
  if( area_ < 0 ) // falls noch nicht berechnet und gecacht
  {
    CalcArea();     // jetzt berechnen
  }
  return area_;
}
```

6. Trotzdem diese Funktion den internen Zustand des Objektes ändert, sollte sie `const` sein. Warum? Da die Funktion nicht den von außen sichtbaren Zustand des Objektes ändert. Wir führen hier ein bisschen Caching durch, was aber ein internes Implementierungsdetail ist und das Objekt dabei logisch `const` bleibt, auch wenn dem physikalisch nicht so ist.

   Folglich sollte die Elementvariable `area_` als `mutable` deklariert werden. Wenn Ihr Compiler `mutable` noch nicht unterstützt, machen Sie Trick 17, benutzten Sie ein `const_cast` für `area_` hier und schreiben Sie einen Kommentar, welcher aussagt, dass die nächste Person den `const_cast` entfernen soll, wenn `mutable` verfügbar ist – aber machen Sie die Funktion `const`.

```
private:
  void InvalidateArea() { area_ = -1; }
```

7. Diese Funktion sollte auch `const` sein, wenn auch aus keinen anderem Grund als Konsistenz. Zugegeben, semantisch wird die Funktion nur von nicht-`const`-Funktionen aufgerufen, da ihr Zweck darin besteht, die gecachte `area_`-Elementvariable ungültig zu machen, wenn sich der Zustand des Objektes ändert. Aber Konsistenz und Klarheit sind wichtig für einen guten Programmierstil, weshalb Sie auch diese Funktion als `const` deklarieren sollten.

```
void CalcArea()
{
  area_ = 0;
  vector<Point>::iterator i;
  for( i = points_.begin(); i != points_.end(); ++i )
    area_ += /* einige Berechnungen */;
}
```

8. Diese Funktion sollte definitiv `const` sein, da sie den von außen sichtbaren Objektzustand nicht verändert. Beachten Sie auch, dass diese Funktion nur von einer anderen `const`-Elementfunktion aus aufgerufen wird, nämlich von `GetArea()`.

9. Der `iterator` sollte den Zustand der `points_`-Menge nicht verändern, so dass ein `const_iterator` verwendet werden sollte.

```
  vector<Point> points_;
  double        area_;
};
Polygon operator+( Polygon& lhs, Polygon& rhs )
{
```

10. Die Parameter sollten selbstverständlich `const`-Referenzen sein.

11. Wiederum sollte bei der Rückgabe via Wert ein `const`-Objekt zurückgegeben werden.

```
  Polygon ret = lhs;
  int last = rhs.GetNumPoints();
```

12. Da `last` sich nie verändern sollte, drücken Sie dies dadurch aus, dass Sie der Variablen den Typ »`const int`« geben.

```
  for( int i = 0; i < last; ++i ) // verknüpfen
  {
    ret.AddPoint( rhs.GetPoint(i) );
  }
  return ret;
}
```

(Haben wir aus dem `rhs`-Parameter eine `const`-Referenz gemacht, sehen wir einen weiteren Grund, warum `GetPoint()` eine `const`-Elementfunktion sein sollte, die entweder einen `const`-Wert zurückgibt oder eine Referenz auf einen `const`-Wert.)

```
void f( const Polygon& poly )
{
  const_cast<Polygon&>(poly).AddPoint( Point(0,0) );
}
```

Bonus: Das Ergebnis des `const_cast` ist nicht definiert, wenn das referenzierte Objekt als `const` deklariert wurde – was für `f(cpoly)` unten der Fall ist. Der Parameter ist nicht wirklich ein `const`-Parameter, weshalb Sie ihn auch nicht als `const` deklarieren sollten. Das kommt einer Lüge gleich.

```
Void g( Polygon& const rPoly ) { rPoly.AddPoint( Point(1,1) ); }
```

13. Dieses `const` ist nicht nur nutzlos, da Referenzen sowieso in dem Sinn konstant sind, als dass sie nicht auf eine andere Variable verbogen werden können (so dass sie auf ein anderes Objekt verweisen), sondern an dieser Stelle auch nicht erlaubt.

```
void h( Polygon* const pPoly ) { pPoly->AddPoint( Point(2,2) ); }
```

14. Diese `const` ist genauso nutzlos, aber aus einem anderen Grund: Da wir den Zeiger als Wert übergeben, macht dies ebenso wenig Sinn wie das Übergeben eines `const` int-Parameters oben. Denken Sie darüber nach – es ist genau die gleiche Sache, da dieses `const` nur verspricht, dass Sie den Zeiger nicht verändern werden. Dies ist kein nützliches Versprechen, da der Aufrufer sich um diese Entscheidung nicht kümmern kann oder davon betroffen ist.

```
int main()
{
  Polygon poly;
  const Polygon cpoly;
  f(poly);
```

Das ist in Ordnung.

```
  f(cpoly);
```

Dies führt zu undefinierten Verhalten, wenn `f()` versucht, die Konstantheit des Parameters wegzucasten und ihn dann zu modifizieren. Siehe oben.

```
  g(poly);
```

Das ist in Ordnung.

```
  h(&poly);
}
```

Das ist auch in Ordnung.

Ihre Freunde const und mutable

Gehen Sie mit `const` richtig um: `const` und `mutable` sind Ihre Freunde.

Ich treffe immer noch Programmierer, die denken, dass `const` den ganzen Ärger nicht wert ist. »Ah, es ist anstrengend, überall `const` schreiben zu müssen«, habe ich einige sich beschweren hören. »Wenn ich es an einer Stelle benutze, muss ich es überall benutzen. Und sowieso, andere Leute benutzen es auch nicht, und deren Programme funktionieren toll. Einige der Bibliotheken, die ich benutze, sind auch nicht `const`-korrekt. Ist es die Mühe überhaupt wert?«

Wir könnten uns ein ähnliches Szenario vorstellen, dieses Mal in Gewehrreichweite: »Ah, es ist anstrengend, diese Gewehrsicherung jedes Mal umzulegen. Und sowieso, einige andere Leute benutzen sie auch nicht, und einige von ihnen haben sich auch noch nicht in ihren Fuß geschossen ...«

Sicherheits-inkorrekte Gewehrmänner leben nicht lange in dieser Welt. So auch `const`-inkorrekte Programmierer, Tischler, die keine Zeit für schlechte Kunden haben, und Elektriker, die keine Zeit haben, das Strom führende Kabel zu identifizieren. Es gibt keine Entschuldigung dafür, die Sicherheitsmechanismen, die zusammen mit einem Produkt ausgeliefert werden, zu ignorieren, und es gibt keine Entschuldigung für Programmierer, so faul zu sein, keinen `const`-richtigen Code zu schreiben. (Vielleicht gar nicht so merkwürdig, die Leute, die `const` weglassen, sind die gleichen, die dazu tendieren, Compiler-Warnungen zu ignorieren. Machen Sie sich das mal klar.) In vielen Entwicklerbuden sind das Weglassen von `const` und das Ignorieren von Compiler-Warnungen »einmalige« Vergehen und das zu Recht.

Wann immer möglich `const` zu verwenden, sorgt für sichereren und klareren Code. Sie können damit Schnittstellen und Invarianten weitaus effektiver dokumentieren, als mit einem `/* Ich verspreche diese Variable nicht zu ändern */`-Kommentar. Es ist ein mächtiger Teil des *»design by contract«* – Entwurf per Vertrag – Prinzips. Es hilft dem Compiler Sie vor aus Versehen falsch geschriebenem Code zu bewahren. Es kann dem Compiler sogar helfen, kleineren oder schnelleren Code zu produzieren. Darum gibt es keinen Grund, warum Sie `const` nicht so oft wie möglich benutzen sollten, sondern nur Gründe dafür.

Erinnern Sie sich, dass der richtige Gebrauch des `mutable`-Schlüsselwortes eine wesentliche Rolle bei der `const`-Korrektheit spielt. Wenn Ihre Klasse eine Elementvariable enthält, die selbst für `const`-Objekte und -Operationen geändert werden könnte, deklarieren Sie diese Elementvariable als `mutable`. Auf diese Weise sind Sie in der Lage, die `const`-Elementfunktionen Ihrer Klasse einfach und richtig zu schreiben, und Benutzer Ihrer Klasse sind in der Lage richtige konstante nicht-konstante Objekte vom Typ Ihrer Klasse zu kreieren.

Es ist wahr, dass nicht alle kommerziellen Bibliotheken `const`-richtig sind. Das ist aber trotzdem keine Entschuldigung für Sie, `const`-unkorrekten Code zu schreiben. (Es ist jedoch eine der wenigen guten Entschuldigungen dafür, `const_cast` zu verwenden.) Schreiben Sie Ihren eigenen Code immer `const`-richtig, und wenn Sie die `const`-falsche Bibliothek aufrufen, können Sie den passenden `const_cast` wann immer nötig verwenden – vorzugsweise mit einem ausführlichen Kommentar über die Faulheit der Bibliotheksprogrammierer und wie sehnsüchtig Sie sich nach einem besseren Ersatz sehnen.

Wenn Sie immer noch Zweifel gegenüber `const` hegen, beeilen Sie sich und lesen Sie nochmals Lektion 21 *»Use `const` whenever possible«* in Meyers98. Es gibt Gründe dafür, dass `const` in C schon vor der Standardisierung eingeführt wurde. Zurück zum Anfang 1980, hat ein Forscher namens Stroutroup die witzige niedlich Sprache »C mit Klassen« erfunden, welche das witzige kleine Schlüsselwort `readonly` enthielt, das nachweislich nützlich zum Spezifizieren klarerer Schnittstellen war, und damit für das Schreiben sichereren Codes. Die C-Leute des Bell Labs mochten die Idee, zogen aber den Namen `const` vor. Als das ANSI-Komitee wenig später ins Rollen kam, adoptierten Sie das gleiche Sprachmerkmal mit genau dem gleichen Namen – und der Rest ist, wie sie es sagen, Geschichte (Stroutrup94, Seite 90).

Schießen Sie sich nicht selbst (wie Ihre Programmier-Kollegen) in die Füße. Sorgen Sie immer für Ihre eigene Sicherheit: Seien Sie `const`-korrekt. Schreiben Sie `const` wann immer möglich und lächeln Sie bei dem Gedanken an klareren, sichereren und besseren Code.

Das war es. Hier die überarbeitete Version des Codes, in dem alle `const`-Stolpersteine aus dem Weg geräumt sind (aber nicht versucht wurde, irgendwelche anderen Stilnoten zu verbessern).

```
class Polygon
  {
  public:
    Polygon() : area_(-1) {}
    void        AddPoint( Point pt ) { InvalidateArea();
                                       points_.push_back(pt); }
    const Point GetPoint( int i ) const  { return points_[i]; }
    int         GetNumPoints() const { return points_.size(); }
    double GetArea() const
    {
      if( area_ < 0 ) // falls noch nicht berechnet und gecacht
      {
        CalcArea();     // jetzt berechnen
      }
      return area_;
    }
  private:
    void InvalidateArea() const { area_ = -1; }
    void CalcArea() const
    {
      area_ = 0;
      vector<Point>::const_iterator i;
      for( i = points_.begin(); i != points_.end(); ++i )
      {
        area_ += /* einige Berechnungen */;
      }
    }
```

```
  vector<Point>  points_;
  mutable double area_;
};
const Polygon operator+( const Polygon& lhs,
                         const Polygon& rhs )
{
  Polygon ret = lhs;
  const int last = rhs.GetNumPoints();
  for( int i = 0; i < last; ++i ) // verknüpfen
  {
    ret.AddPoint( rhs.GetPoint(i) );
  }
  return ret;
}
void f( Polygon& poly )
{
  poly.AddPoint( Point(0,0) );
}

void g( Polygon& rPoly ) { rPoly.AddPoint( Point(1,1) ); }

void h( Polygon* pPoly ) { pPoly->AddPoint( Point(2,2) ); }

int main()
{
  Polygon poly;
  f(poly);
  g(poly);
  h(&poly);
}
```

Lektion 44: Casts — Schwierigkeitsgrad: 6

Wie gut kennen Sie C++-Casts? Sie richtig einzusetzen kann die Zuverlässigkeit Ihres Codes drastisch verbessern.

Die neuen Casts in Standard-C++ bieten mehr Macht und Sicherheit als die alten Casts im C-Stil. Wie gut kennen Sie sie? Der Rest dieses Problems benutzt die folgenden Klassen und globalen Variablen:

```
class  A { public: virtual ~A(); /*...*/ };
A::~A() { }
class B : private virtual A  { /*...*/ };
class C : public  A          { /*...*/ };
class D : public B, public C { /*...*/ };
A a1; B b1; C c1; D d1;
const A  a2;
const A& ra1 = a1;
const A& ra2 = a2;
char c;
```

Diese Lektion besteht aus vier Fragen:

1. Welcher der folgenden Casts im neuen Still ist ***nicht*** äquivalent zu einem alten Cast im C-Stil?

```
const_cast
dynamic_cast
reinterpret_cast
static_cast
```

2. Schreiben Sie für jeden der folgenden C-Stil-Casts den äquivalenten Cast im neuen Stil. Welche Casts sind falsch, wenn sie nicht im neuen Stil geschrieben werden?

```
void f()
{
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
  pa = (A*)&a2;
  pb = (B*)&c1;
  pc = (C*)&d1;
}
```

3. Kritisieren Sie jeden der folgenden C++-Casts hinsichtlich des Stils und der Richtigkeit.

```
void g()
{
  unsigned char* puc = static_cast<unsigned char*>(&c);
  signed char* psc = static_cast<signed char*>(&c);
  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
  B* pb2 = static_cast<B*>(&b1);
  A* pa1 = const_cast<A*>(&ra1);
  A* pa2 = const_cast<A*>(&ra2);
  B* pb3 = dynamic_cast<B*>(&c1);
  A* pa3 = dynamic_cast<A*>(&b1);
  B* pb4 = static_cast<B*>(&d1);
  D* pd = static_cast<D*>(pb4);
  pa1 = dynamic_cast<A*>(pb2);
  pa1 = dynamic_cast<A*>(pb4);
  C* pc1 = dynamic_cast<C*>(pb4);
  C& rc1 = dynamic_cast<C&>(*pb2);
}
```

4. Warum ist es typischerweise nicht sinnvoll, eine nicht-`const`-Variable via `const_cast` in eine `const`-Variable zu casten? Demonstrieren Sie anhand eines gültigen Beispiels, wann es doch sinnvoll sein kann.

Lösung

Lassen Sie uns die Fragen eine nach der anderen beantworten.

1. Welcher der folgenden Casts im neuen Still ist *nicht* äquivalent zu einem alten Cast im C-Stil?

   Nur `dynamic_cast` ist nicht äquivalent zu alten Casts im C-Stil. Alle anderen Casts im neuen Stil haben äquivalente C-Stil-Casts.

☑ Richtlinie

Bevorzugen Sie Casts im neuen Stil.

2. Schreiben Sie für jeden der folgenden C-Stil-Casts den äquivalenten Cast im neuen Stil. Welche Casts sind falsch, wenn sie nicht im neuen Stil geschrieben werden?

```
void f()
{
  A* pa; B* pb; C* pc;
  pa = (A*)&ra1;
```

   Benutzen Sie stattdessen einen `const_cast`:

```
  pa = const_cast<A*>(&ra1);
  pa = (A*)&a2;
```

   Dies kann nicht als Cast im neuen Stil ausgedrückt werden. Der beste Kandidat wäre `const_cast`, da aber `a2` ein `const`-Objekt ist, ist das Ergebnis nicht definiert.

```
  pb = (B*)&c1;
```

   Benutzen Sie stattdessen einen `reinterpret_cast`:

```
  pb = reinterpret_cast<B*>(&c1);
  pc = (C*)&d1;
}
```

   Der obige Cast ist in C nicht zulässig. In C++ ist hingegen kein Cast erforderlich:

```
  pc = &d1;
```

3. Kritisieren Sie jeden der folgenden C++-Casts hinsichtlich des Stils und der Richtigkeit.

   Zunächst eine allgemeine Bemerkung: Alle der folgenden `dynamic_cast`-Ausdrücke wären falsch, wenn die involvierten Klassen keine virtuellen Funktionen hätten. Zum Glück stellt `A` eine virtuelle Funktion zur Verfügung, so dass alle `dynamic_cast` zulässig sind.

```
void g()
{
  unsigned char* puc = static_cast<unsigned char*>(&c);
  signed char* psc = static_cast<signed char*>(&c);
```

Falsch: Wir müssen in beiden Fällen `reinterpret_cast` benutzen. Dies mag Sie zunächst überraschen, aber der Grund hierfür ist, dass `char`, `signed char` und `unsigned char` drei verschiedene Typen sind. Auch wenn sie implizit ineinander umgewandelt werden können, sind sie nicht verwandt, so dass auch Zeiger auf sie nicht verwandt sind.

```
  void* pv = static_cast<void*>(&b1);
  B* pb1 = static_cast<B*>(pv);
```

Diese beiden Casts sind in Ordnung, allerdings ist der erste unnötig, da es bereits eine implizite Umwandlung von Zeiger auf irgendeinen Datentyp zu einem `void*`-Zeiger gibt.

```
  B* pb2 = static_cast<B*>(&b1);
```

Das ist in Ordnung, aber überflüssig, da das Argument bereits vom Typ `B*` ist.

```
  A* pa1 = const_cast<A*>(&ra1);
```

Das ist erlaubt, allerdings deutet das Hinwegcasten von `const` (oder `volatile`) auf einen schlechten Programmierstil hin. Die meisten Fälle, in denen Sie legitim die Konstantheit eines Zeigers oder einer Referenz wegcasten wollen, treten im Zusammenhang mit Elementvariablen auf und werden durch das `mutable`-Schlüsselwort abgedeckt. Siehe Lektion 43 für eine weiterführende Diskussion über `const`-Richtigkeit.

☑ Richtlinie

Vermeiden Sie es, `const` wegzucasten. Benutzen Sie stattdessen `mutable`.

```
  A* pa2 = const_cast<A*>(&ra2);
```

Dies ist ein Fehler: Dies führt zu undefiniertem Verhalten, wenn der Zeiger für Schreiboperationen auf das Objekt benutzt wird, da `a2` wirklich ein `const`-Objekt ist. Um zu sehen, warum es sich hierbei um ein echtes Problem handelt, beachten Sie, dass es dem Compiler möglich ist, zu sehen, dass `a2` als `const`-Objekt erzeugt wurde und er diese Information benutzten kann, um das Objekt aus Optimierungsgründen in Readonly-Speicher abzulegen. `const` für solche Objekte wegzucasten ist extrem gefährlich.

☑ Richtlinie

Vermeiden Sie es, `const` wegzucasten.

```
B* pb3 = dynamic_cast<B*>(&c1);
```

Potenzieller Fehler (wenn Sie versuchen, `pb3` zu benutzen): Da `c1` KEIN `B` IST (weil `C` nicht öffentlich von `B` abgeleitet ist – hier ist `C` sogar überhaupt nicht von `B` abgeleitet), führt diese Anweisung dazu, dass `pb3` auf null gesetzt wird. Der einzig erlaubte Cast wäre ein `reinterpret_cast`, und diesen zu benutzen ist fast immer teuflisch.

```
A* pa3 = dynamic_cast<A*>(&b1);
```

Wahrscheinlich ein Fehler: Da `b1` KEIN `A` IST (weil `B` nicht öffentlich von `A` abgeleitet ist; die Ableitung ist `private`), ist dies illegal, falls `g()` kein `friend` von `B` ist.

```
B* pb4 = static_cast<B*>(&d1);
```

Das ist in Ordnung, aber unnötig, da die Umwandlung eines Zeigers auf eine abgeleitete Klasse in einen Zeiger auf eine Basisklasse implizit durchgeführt werden kann.

```
D* pd = static_cast<D*>(pb4);
```

Das ist in Ordnung, was Sie vielleicht überrascht, wenn Sie gedacht haben, dass dies einen `dynamic_cast` erfordert. Der Grund hierfür ist, dass Downcasts auch mit `static_cast` durchgeführt werden können, wenn das Zielobjekt bekannt ist, aber Vorsicht: Sie teilen dem Compiler so mit, dass Sie ganz genau wissen, dass das, worauf gezeigt wird, von dem angegebenen Typ ist. Wenn Sie sich irren, kann der Cast Sie nicht über dieses Problem informieren (wie dies bei `dynamic_cast` der Fall wäre, wo ein Null-Zeiger zurückgegeben wird, wenn der Cast fehlschlägt) und bestenfalls erhalten Sie unechte Laufzeit-Fehler und/oder Programmabstürze.

☑ Richtlinie

Vermeiden Sie Downcasts.

```
pa1 = dynamic_cast<A*>(pb2);
pa1 = dynamic_cast<A*>(pb4);
```

Diese beiden Zeilen sehen sich sehr ähnlich. In beiden wird versucht einen `dynamic_cast` zu benutzen, um einen `B*`-Zeiger in einen `A*`-Zeiger zu casten. Wie dem auch sei, stellt die erste Zeile einen Fehler dar, die zweite jedoch nicht.

Hier der Grund: Wie oben bereits bemerkt, können Sie `dynamic_cast` nicht dafür benutzen, um einen Zeiger, der wirklich auf ein `B`-Objekt zeigt (und `pb2` zeigt hier auf das Objekt `b1`), in einen Zeiger zu casten, der auf ein `A`-Objekt zeigt, da `B` `private`

und nicht `public` von `A` abgeleitet ist. Wie dem auch sei, der zweite Cast ist hingegen erfolgreich, da `pb4` auf das Objekt `d1` zeigt, `A` für `D` eine indirekte öffentliche Basisklasse (über Klasse `C`) ist und `dynamic_cast` in der Lage ist, den Cast mit Hilfe des Pfades `B*` à `D*` à `C*` à `A*` in der Vererbungshierarchie durchzuführen.

```
C* pc1 = dynamic_cast<C*>(pb4);
```

Dies ist aus dem gleichen Grund in Ordnung: `dynamic_cast` kann die Vererbungshierarchie entlang navigieren, um Cross-Casts auszuführen, so dass auch diese Anweisung zulässig ist und erfolgreich ausgeführt wird.

```
  C& rc1 = dynamic_cast<C&>(*pb2);
}
```

Zum Schluss ein »Exception-Fehler«: Da `*pb2` nicht wirklich ein `C`-Objekt ist, wirft `dynamic_cast` die Exception `bad_cast`, um das Fehlschlagen des Casts zu signalisieren. Warum? Nun, dynamic_cast kann und wird einen Null-Zeiger zurückliefern, wenn ein Zeiger-Cast fehlschlägt, da es aber nichts wie eine Null-Referenz gibt, kann auch keine Null-Referenz zurückgeliefert werden, wenn ein Referenz-Cast fehlschlägt. Es gibt keine Möglichkeit solch einen Fehlschlag dem Client-Code anzuzeigen, außer dem Werfen einer Exception, und genau hierfür ist die Standard-Exception-Klasse `bad_cast` da.

4. Warum ist es typischerweise nicht sinnvoll, eine nicht-`const`-Variable via `const_cast` in eine `const`-Variable zu casten?

   Die ersten drei Fragen enthielten kein Beispiel für die Benutzung von `const_cast`, um Konstantheit hinzuzufügen, zum Beispiel um einen Zeiger auf ein nicht-`const`-Objekt in einen Zeiger auf ein `const`-Objekt zu konvertieren. Schließlich ist das explizite Hinzufügen von Konstantheit gewöhnlich redundant – zum Beispiel ist es erlaubt, einem Zeiger, der auf ein nicht-`const`-Objekt verweist, einen Zeiger zuzuweisen, der auf ein `const`-Objekt verweist. Normalerweise benötigen wir `const_cast` nur für die umgekehrte Richtung.

   Der letzte Teil der Frage war: Demonstrieren Sie anhand eines gültigen Beispiels, wann es doch sinnvoll sein kann.

   Es gibt mindestens einen Fall, in dem `const_cast` sinnvoll ist, um von einer nicht-`const`-Variable in eine `const`-Variable zu casten – um eine spezielle überladene Funktion aufzurufen oder eine spezifische Version eines Templates zu instanziieren. Zum Beispiel:

```
void f( T& );
void f( const T& );
template<class T> void g( T& t )
{
  f( t );                        // calls f(T&)
  f( const_cast<const T&>(t) ); // calls f(const T&)
}
```

Natürlich ist es gewöhnlich einfacher, wenn eine spezifische Version eines Templates ausgewählt werden soll, diese explizit zu benennen, anstatt die richtige Erkennung zu erzwingen. Zum Beispiel um die richtige Version einer Template-Funktion `h()` aufzurufen, sollten Sie besser »`h<const T&>( t )`« statt »`h( const_cast<const T&> (t) )`« schreiben.

Lektion 45: `bool` **Schwierigkeitsgrad: 7**

Benötigen wir wirklich einen eingebauten `bool`-Typ? Warum emulieren wir ihn nicht einfach mit den existierenden Sprachmöglichkeiten? Diese Lektion enthüllt die Antwort.

Neben `wchar_t` (was ein `typedef` in C ist) ist `bool` der einzige eingebaute Typ, der zu C++ seit dem ARM (Ellis90) hinzugefügt wurde.

Hätten die Effekte von `bool` dupliziert werden können, ohne einen zusätzlich eingebauten Datentyp einzuführen? Wenn ja, zeigen Sie eine äquivalente Implementierung. Wenn nein, zeigen Sie, warum mögliche Implementierungen sich nicht so verhalten wie der eingebaute `bool`-Typ.

Lösung

Die Antwort ist: Nein, die Effekte von `bool` hätten nicht dupliziert werden können, ohne einen zusätzlich eingebauten Datentyp einzuführen. Der eingebaute Datentyp `bool` sowie die Schlüsselwörter `true` und `false` wurden genau aus dem Grund zu C++ hinzugefügt, weil sie nicht vollständig durch die existierende Sprache dupliziert werden konnten.

Diese Lektion soll dazu dienen zu illustrieren, was Sie beachten müssen, wenn Sie eigene Klassen, `enum`s oder andere Hilfsmittel entwerfen.

Der zweite Teil der Frage zu dieser Lektion war: Wenn nein, zeigen Sie, warum mögliche Implementierungen sich nicht so verhalten, wie der eingebaute `bool`-Typ.

Es gibt vier oft gesehene Implementierungen.

Möglichkeit 1: typedef (8,5 / 10 Punkte)

Diese Möglichkeit bedeutet »`typedef <irgendwas> bool;`«, typischerweise:

```
// Möglichkeit 1: typedef
//
typedef int bool;
const bool true  = 1;
const bool false = 0;
```

Diese Lösung ist nicht schlecht, erlaubt jedoch kein Überladen für `bool`. Zum Beispiel:

```
// Datei f.h
void f( int  ); // ok
void f( bool ); // ok, neue Deklaration der gleichen Funktion

// Datei f.cpp
void f ( int  ) { /*...*/ } // ok
void f ( bool ) { /*...*/ } // Fehler, Neudefinition
```

Ein weiteres Problem mit Möglichkeit 1 ist, dass Code wie folgender zu unerwartetem Verhalten führen kann:

```
void f( bool b )
{
  assert( b != true && b != false );
}
```

Also ist Möglichkeit 1 nicht gut genug.

Möglichkeit 2: #define (0 / 10 Punkte)

Diese Möglichkeit bedeutet »`#defines bool <irgendwas>`«, typischerweise:

```
// Möglichkeit 2: #define
//
#define bool int
#define true  1
#define false 0
```

Dies ist natürlich fürchterlich. Diese Möglichkeit hat nicht nur alle Probleme, die auch Möglichkeit 1 hat, sondern leidet auch unter den typischen Verwüstungen von `#define`-Direktiven. Zum Beispiel sei da der arme Programmierer genannt, der versucht diese Bibliothek zu benutzen und schon eine Variable mit dem Namen `false` hat; dies führt definitiv zu einem anderen Verhalten als mit einem eingebauten Typ.

Zu versuchen den Präprozessor zu verwenden, um einen Datentyp zu simulieren, ist einfach eine schlechte Idee.

Möglichkeit 3: enum (9 / 10 Punkte)

Diese Möglichkeit bedeutet eine Aufzählung »`enum bool`« zu definieren, typischerweise:

```
// Möglichkeit 3: enum
//
enum bool { false, true };
```

Dies ist etwas besser als Möglichkeit 1, da nun Überladen möglich ist (das Hauptproblem von Möglichkeit 1), jedoch ist keine automatische Umwandlung von einem Konditionalausdruck mehr möglich (was mit Möglichkeit 1 gegangen wäre), wie im folgenden Beispiel:

```
bool b;
b = ( i == j);
```

Dies funktioniert nicht, da ein `int` nicht automatisch in einen `enum` konvertiert werden kann.

Möglichkeit 4: class (9 / 10 Punkte)

Hey, das ist doch eine objekt-orientierte Sprache, oder? Warum schreiben wir also keine Klasse, typischerweise:

```
// Möglichkeit 4: class
//
class bool
{
public:
  bool();
  bool( int );              // um Umwandlung von Konditional-
  bool& operator= ( int ); // ausdrücken zu ermöglichen
  //operator int();   // fragwürdig!
  //operator void*(); // fragwürdig!
private:
  unsigned char b_;
}
const bool true ( 1 );
const bool false( 0 );
```

Dies funktioniert ausgenommen für die Umwandlungsoperatoren, die als fragwürdig markiert wurden. Sie sind aus folgenden Gründen fragwürdig:

- Mit einer automatischen Konvertierung würde `bool` mit der Auflösung bei Überladung interferieren, wie es auch bei allen Klassen mit nicht-expliziten (Konvertierungs-)Konstruktoren und/oder impliziten Umwandlungsoperatoren der Fall ist, insbesondere bei Umwandlungen von einem oder in einen verbreiteten Typ. Siehe Lektion 20 und 39 für mehr Informationen zu implizierter Umwandlung.
- Ohne eine Umwandlung in etwas wie `int` oder `void*` können `bool`-Objekte nicht auf »natürliche« Art und Weise in Bedingungen benutzt werden. Zum Beispiel:

```
bool b;
/*...*/
if( b ) // Fehler ohne automatische Umwandlung
```

```
{       // in etwas wie int oder void*
  /*...*/
}
```

Somit stecken wir in einer klassischen Zwickmühle: Wir müssen die eine oder die andere Alternative wählen – entweder wir stellen eine automatische Umwandlung zur Verfügung oder nicht – jedoch können wir mit keiner der beiden Alternativen den Effekt eines eingebauten `bool`-Typs duplizieren. Zusammenfassend kann gesagt werden:

- `typedef ... bool` würde kein Überladen für `bool` erlauben.
- `#define bool ...` würde auch kein Überladen erlauben und uns zusätzlich die typischen Fallen von #define-Direktiven bescheren.
- `enum bool ...` würde das Überladen zulassen, jedoch wäre keine automatische Umwandlung von Konditionalausdrücken (wie in »`b = ( i == j);`«) mehr möglich.
- `class bool ...` würde das Überladen erlauben, aber es könnte kein einzelnes `bool`-Objekt in Bedingungen (wie »`if( b )`«) getestet werden, falls nicht eine automatische Umwandlung in etwas wie `int` oder `void*` zur Verfügung gestellt wird, was aber wiederum zu den bekannten Schwierigkeiten beim Überladen führen würde.

Schließlich gibt es noch etwas (im Zusammenhang mit Überladen), das wir nicht anders hätten machen können, außer vielleicht mit der letzten Möglichkeit: Festzulegen, dass Konditionalausdrücke den Typ `bool` haben.

Somit benötigten wir in der Tat einen eingebauten `bool`-Typ.

Lektion 46: Vorwärtsfunktionen — Schwierigkeitsgrad: 3

Wie wird eine Vorwärtsfunktion (forwarding function) am besten geschrieben? Die grundlegende Antwort auf diese Frage ist einfach, aber wir werden in dieser Lektion auch eine subtile Änderung der Sprache kennen lernen, die erst kurz vor der Verabschiedung des Standards gemacht wurde.

Vorwärtsfunktionen sind nützliche Hilfsmittel, um Arbeit an andere Funktionen oder Objekte abzugeben, insbesondere, wenn die Abgabe effektiv durchgeführt wird.

Kritisieren Sie die folgende Vorwärtsfunktion. Würden Sie sie ändern? Wenn ja, wie?

```
// Datei f.cpp
//
#include "f.h"
/*...*/
bool f( X x )
```

```
{
  return g( x );
}
```

Lösung

Erinnern Sie sich an die Einleitung vor der Frage? Sie lautete: Vorwärtsfunktionen sind nützliche Hilfsmittel, um Arbeit an andere Funktionen oder Objekte abzugeben, insbesondere, wenn die Abgabe effektiv durchgeführt wird.

Diese Einleitung trifft ins Herz des Problems – Effizienz.

Es gibt zwei wesentliche Verbesserungen, die diese Funktion effizienter machen würden. Die erste sollte immer gemacht werden; die zweite ist manchmal sinnvoll.

1. Übergeben Sie den Parameter als `const&` und nicht via Wert. »Ist das nicht total offensichtlich?«, mögen Sie sich fragen. Nein, ist es nicht, nicht in diesem speziellen Fall. Bis vor kurzem – 1997 – besagte der C++-Draft-Standard, weil Compiler erkennen können, dass ein Parameter `x` niemals für etwas anderes gebraucht wird, als dass er an `g()`übergeben wird, kann der Compiler sich dazu entschließen, `x` vollständig zu verwerfen (das heißt, x als unnötig zu eliminieren). Wenn zum Beispiel der Client-Code so aussieht:

   ```
   X my_x;
   f( my_x );
   ```

   dann war es bis dahin dem Compiler erlaubt, eines der beiden folgenden Dinge zu tun:

   - Eine Kopie von `my_x` für `f()` zu erzeugen (dies wäre der Parameter namens `x` im Sichtbarkeitsbereich von `f()`) und diese Kopie an `g()` zu übergeben.
   - Oder `my_x` direkt an `g()` zu übergeben, ohne überhaupt eine Kopie zu erzeugen, da der Compiler erkennt, dass die zusätzliche Kopie nie zu etwas anderem gebraucht wird, als sie als Parameter an `g()` zu übergeben.

   Die zweite Möglichkeit ist sehr schön effizient, nicht wahr? Das ist es, wofür optimierende Compiler da sind, oder?

   Ja und ja, bis zum Treffen des Standardisierungskomitees 1997 in London. Bei diesem Treffen wurde der Draft-Standard so verbessert, dass größere Restriktionen für die Situationen festgelegt wurden, in denen es Compilern erlaubt ist, »zusätzliche« Kopien zu verwerfen. Diese Änderung war nötig, um Probleme zu vermeiden, die auftreten, wenn Compilern erlaubt ist, Copy-Konstruktion zu verwerfen, insbesondere wenn diese Seiteneffekte hat. Es gibt einige einsichtige Fälle, in denen gültiger Code von der Anzahl der gemachten Kopien eines Objektes abhängen kann.

Heutzutage ist die einzige Situation, in der ein Compiler immer noch Copy-Konstruktion verwerfen kann, die bei der Optimierung von Rückgabewerten und temporären Objekten (die Details lesen Sie in Ihrem Lieblings-C++-Buch nach). Das bedeutet, dass für Vorwärtsfunktionen wie `f()` der Compiler nun dazu verpflichtet ist, zwei Kopien anzufertigen. Da wir (als die Autoren von `f()`) wissen, dass in diesem Fall die zusätzliche Kopie nicht notwendig ist, sollten wir uns auf unsere allgemeine Regel rückbesinnen und `x` als `const X&`-Parameter deklarieren.

☑ Richtlinie

Bevorzugen Sie die Übergabe via Referenz statt via Wert, benutzen Sie const wann immer möglich.

Beachten Sie: Wenn wir der allgemeinen Regel von Anfang an gefolgt wären, anstatt zu versuchen Vorteile aus dem weit reichenden Wissen darüber zu ziehen, was dem Compiler zu tun erlaubt ist, hätten uns die Änderungen am Standard gar nicht betroffen. Dies ist ein wunderbares Beispiel dafür, warum einfach oft besser ist – vermeiden Sie die dunklen Ecken einer Programmiersprache, so oft Sie können, und versuchen Sie nicht, von schlauen Feinheiten der Sprachspezifikation abhängig zu sein.

☑ Richtlinie

Vermeiden Sie die »dunklen Ecken« einer Programmiersprache; benutzen Sie die einfachsten Techniken, die effektiv sind.

2. Machen Sie die Funktion `inline`. Dies ist manchmal sinnvoll. Kurz gesagt, bevorzugen Sie es, zunächst alle Funktionen ohne `inline` zu schreiben, und deklarieren Sie dann sukzessive individuelle Funktionen falls nötig als `inline`, wenn der Performance-Vorteil hierdurch wirklich von Bedeutung ist.

☑ Richtlinie

Vermeiden Sie `inline`-Funktionen und weiter gehendes Tuning, bis Perfomance-Profiling gezeigt hat, dass dies wirklich notwendig ist.

Wenn Sie die Funktion als `inline` deklarieren, ist der positive Effekt der, dass Sie sich den Overhead des Funktionsaufrufs von `f()` sparen.

Der negative Nebeneffekt ist, dass, wenn `f()` als `inline` deklariert wird, die Implementierung von `f()` offen gelegt wird und Client-Code davon abhängig wird, so dass, falls

sich `f()` ändert, aller Client-Code neu kompiliert werden muss. Schlimmer noch, Client-Code benötigt zumindest einen Prototyp für die Funktion `g()`, was ein bisschen beschämend ist, da Client-Code eigentlich `g()` nie direkt aufruft und wahrscheinlich den Prototyp von `g()` vorher nie gebraucht hat (zumindest nicht, soweit wir aus dem Beispiel erkennen können). Und wenn `g()` selbst geändert wird, so dass die Funktion andere Parameter noch anderer Typen erwartet, hängt der Client-Code auch noch von diesen Klassendeklarationen ab.

Sowohl `inline`-Deklarationen als auch der Verzicht auf `inline` können sinnvoll sein. Die Entscheidung und die damit einhergehenden Vor- und Nachteile hängen davon ab, was (und wie viel) Sie über die heutige Benutzung von `f()` wissen und wie (und wie oft) eine Änderung an `f()` wahrscheinlich ist.

Lektion 47: Kontrollfluss — Schwierigkeitsgrad: 6

Wie gut kennen Sie die Reihenfolge, in der C++-Code ausgeführt wird, wirklich? Testen Sie Ihr Wissen mit folgendem Problem.

»Der Teufel steckt im Detail.« Finden Sie so viele Probleme wie möglich im folgenden (stellenweise erfundenen) Code, insbesondere solche, die etwas mit dem Kontrollfluss zu tun haben.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

//  Die folgenden Zeilen stammen aus einer anderen Headerdatei
//
char* itoa( int value, char* workArea, int radix );
extern int fileIdCounter;

//  Hilfsfunktion/-Klasse zum Automatisieren
//  der Überprüfung der Invarianten einer Klasse
//
template<class T>
inline void AAssert( T& p )
{
  static int localFileId = ++fileIdCounter;
  if( !p.Invariant() )
  {
    cerr << "Invariant failed: file " << localFileId
         << ", " << typeid(p).name()
         << " at " << static_cast<void*>(&p) << endl;
    assert( false );
  }
```

```
}

template<class T>
class AInvariant
{
public:
  AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
  ~AInvariant()              { AAssert( p_ ); }
private:
  T& p_;
};
#define AINVARIANT_GUARD AInvariant<AIType> invariantChecker( *this )

//--------------------------------------------------------------
template<class T>
class Array : private ArrayBase, public Container
{
  typedef Array AIType;
public:
  Array( size_t startingSize = 10 )
  : Container( startingSize ),
    ArrayBase( Container::GetType() ),
    used_(0),
    size_(startingSize),
    buffer_(new T[size_])
  {
    AINVARIANT_GUARD;
  }

  void Resize( size_t newSize )
  {
    AINVARIANT_GUARD;
    T* oldBuffer = buffer_;
    buffer_ = new T[newSize];
    copy( oldBuffer, oldBuffer+min(size_,newSize), buffer_ );
    delete[] oldBuffer;
    size_ = newSize;
  }

  string PrintSizes()
  {
    AINVARIANT_GUARD;
    char buf[30];
    return string("size = ") + itoa(size_,buf,10) +
           ", used = " + itoa(used_,buf,10);
  }

  bool Invariant()
  {
    if( used_ > 0.9*size_ ) Resize( 2*size_ );
    return used_ <= size_;
```

```
  }
private:
  T*     buffer_;
  size_t used_, size_;
};

int f( int& x, int y = x ) { return x += y; }
int g( int& x )            { return x /= 2; }

int main( int, char*[] )
{
  int i = 42;
  cout << "f(" << i << ") = " << f(i) << ", "
       << "g(" << i << ") = " << g(i) << endl;
  Array<char> a(20);
  cout << a.PrintSizes() << endl;
}
```

Lösung

»Löwen und Tiger und Bären, oh Gott!« – Dorothy

Verglichen mit den Fehlern im Code dieser Lektion, kann sich Dorothy über nichts beschweren. Lassen Sie uns den Code Zeile für Zeile betrachten.

```
#include <cassert>
#include <iostream>
#include <typeinfo>
#include <string>
using namespace std;

//  Die folgenden Zeilen stammen aus einer anderen Headerdatei
//
char* itoa( int value, char* workArea, int radix );
extern int fileIdCounter;
```

Die Anwesenheit einer globalen Variablen sollte uns sofort nach Client-Code auf die Suche gehen lassen, der diese zu benutzen versucht, bevor sie initialisiert ist. Die Reihenfolge der Initialisierung globaler Variablen (einschließlich der statischen Elementvariablen von Klassen) zwischen verschiedenen Übersetzungseinheiten ist nicht definiert.

☑ Richtlinie

Vermeiden Sie den Einsatz globaler oder statischer Objekte. Wenn Sie globale oder statische Objekte benutzen müssen, seien Sie immer sehr vorsichtig hinsichtlich der Initialisierungsreihenfolge.

```
//  Hilfsfunktion/-Klasse zum Automatisieren
//  der Überprüfung der Invarianten einer Klasse
//
template<class T>
inline void AAssert( T& p )
{
  static int localFileId = ++fileIdCounter;
```

Aha! Und hier haben wir einen solchen Fall. Angenommen die Definition von `fileIdCounter` sieht so aus:

```
int fileIdCounter = 100; // beginne Zählung bei 100
```

Wenn der Compiler `fileIdCounter` initialisiert, bevor er irgendein `AAsert<T>::localFileId` initialisiert, ist alles schön und gut – `localFileId` wird der erwartete Wert zugewiesen. Andernfalls wird `localFileId` basierend auf den Wert von `fileIdCounter` vor dessen Initialisierung gesetzt – nämlich `0` für eingebaute Typen, und `localFileId` hat letztendlich einen Wert, der um 100 kleiner als erwartet ist.

```
  if( !p.Invariant() )
  {
    cerr << "Invariant failed: file " << localFileId
         << ", " << typeid(p).name()
         << " at " << static_cast<void*>(&p) << endl;
    assert( false );
  }
}

template<class T>
class AInvariant
{
public:
  AInvariant( T& p ) : p_(p) { AAssert( p_ ); }
  ~AInvariant()              { AAssert( p_ ); }
private:
  T& p_;
};
#define AINVARIANT_GUARD AInvariant<AIType> invariantChecker( *this )
```

Diese Hilfsfunktion und Hilfsklasse sind eine interessante Idee, bei der für jede Client-Klasse, die automatisch ihre Invarianten vor und nach einem Funktionsaufruf überprüfen soll, einfach ein `typedef` auf `AIType` für die Klasse selbst definiert und dann `AINVARIANT_GUARD` als erste Zeile jeder Elementfunktion geschrieben werden muss. Eigentlich keine ganz schlechte Idee.

Im Client-Code weiter unten geht die Idee aber leider daneben. Der Hauptgrund dafür ist, dass `AInvariant` Aufrufe von `assert()` kapselt, die automatisch vom Compiler entfernt werden, wenn das Programm nicht im Debug-Modus kompiliert wird. Der fol-

gende Client-Code wurde wahrscheinlich von einem Programmierer geschrieben, der sich dieser Zusammenhänge und den daraus resultierenden Änderungen der Seiteneffekte nicht bewusst war.

```
//------------------------------------------------------------
template<class T>
class Array : private ArrayBase, public Container
{
  typedef Array AIType;
public:
  Array( size_t startingSize = 10 )
  : Container( startingSize ),
    ArrayBase( Container::GetType() ),
```

In dieser Initialisierungsliste für den Konstruktor stecken zwei potenzielle Probleme. Das erste ist nicht notwendigerweise ein Fehler, wurde aber als Kuriosität drin belassen.

1. Wenn `GetType()` eine statische Elementfunktion ist oder eine Elementfunktion, die von ihrem `this`-Zeiger keinen Gebrauch macht (also keine Elementvariablen benutzt) und sich auch nicht auf irgendwelche Seiteneffekte von Konstruktionen verlässt (zum Beispiel ein statischer Benutzungszähler), dann ist dies nur schlechter Programmierstil, wird aber korrekt ausgeführt.

2. Anderfalls (hauptsächlich wenn `GetType()` eine normale nicht-statische Elementfunktion ist) haben wir ein Problem. Nicht-virtuelle Basisklassen werden von links nach rechts in der Reihenfolge ihrer Deklaration initialisiert, so dass `ArrayBase` vor `Container` initialisiert wird. Leider bedeutet das, dass wir versuchen, eine Elementfunktion des noch nicht initialisierten Unterbasisobjekts `Container` zu benutzen.

☑ Richtlinie

Führen Sie Basisklassen in der Initialisierungsliste eines Konstruktors immer in der gleichen Reihenfolge auf, in der sie bei der Klassendefinition erscheinen.

```
    used_(0),
    size_(startingSize),
    buffer_(new T[size_ ])
```

Dies ist ein ernster Fehler, da die Variablen in Wirklichkeit in der Reihenfolge initialisiert werden, in der sie in der Klassendefinition erscheinen:

```
    buffer_(new T[size_ ])
    used_(0),
    size_(startingSize),
```

So geschrieben wird der Fehler offensichtlich. Der Aufruf von `new[]` alloziert einen Speicherbereich unbekannter Größe – typischerweise null oder ein extrem großer Wert,

je nach dem, ob der Compiler den Speicherplatz des Objekts mit Nullen initialisiert, bevor der Konstruktor aufgerufen wird. Wie dem auch sei, es ist sehr unwahrscheinlich, dass anfangs wirklich `startingSize` Bytes alloziert werden.

☑ Richtlinie

Führen Sie Elementvariablen in der Initialisierungsliste eines Konstruktors immer in der gleichen Reihenfolge auf, in der sie bei der Klassendefinition erscheinen.

```
{
  AINVARIANT_GUARD;
}
```

Hier gibt es eine kleine Effizienzschwäche: Die `Invariant()`-Funktion wird unnötigerweise zweimal aufgerufen, einmal während der Konstruktion und ein zweites Mal während der Destruktion der versteckten temporären Variablen. Das dürfte aber nicht wirklich eine Rolle spielen.

```
void Resize( size_t newSize )
{
  AINVARIANT_GUARD;
  T* oldBuffer = buffer_;
  buffer_ = new T[newSize];
  copy( oldBuffer, oldBuffer+min(size_,newSize), buffer_ );
  delete[] oldBuffer;
  size_ = newSize;
}
```

Hier gibt es ein Kontrollfluss-Problem. Bevor Sie weiterlesen, untersuchen Sie die Funktion nochmals, um zu sehen, ob Sie es finden können. (Ein kleiner Tipp: Es ist sehr offensichtlich.)

Die Antwort ist: Diese Funktion ist nicht exception-sicher. Wenn der Aufruf von `new[]` eine `bad_alloc`-Exception wirft, passiert nichts Schlimmes; es entsteht kein Speicherloch in diesem besonderen Fall. Wenn jedoch ein Copy-Zuweisungsoperator von `T` eine Exception wirft (im Verlauf der `copy`-Operation), dann wird nicht nur das aktuelle Objekt in einem ungültigen Zustand belassen, sondern der ursprüngliche Buffer wird ein Speicherloch, da alle Zeiger auf ihn verloren sind und er so niemals mehr gelöscht werden kann.

☑ Richtlinie

Bemühen Sie sich immer exception-sicheren Code zu schreiben. Strukturieren Sie Ihren Code immer so, dass Ressourcen korrekt wieder freigegeben werden und Daten sich in einem konsistenten Zustand befinden, selbst wenn eine Exception auftritt.

```
string PrintSizes()
{
  AINVARIANT_GUARD;
  char buf[30];
  return string("size = ") + itoa(size_,buf,10) +
         ", used = " + itoa(used_,buf,10);
}
```

Die `itoa`-Funktion, deren Prototyp ganz am Anfang gezeigt wurde, benutzt den übergebenen Buffer als Arbeitsbereich. Dabei kommt es zu einem Kontrollfluss-Problem. Es gibt keinen Weg, die Reihenfolge vorauszusagen, in der die Ausdrücke in der letzten Zeile ausgewertet werden, da die Reihenfolge, in der Funktionsargumente ausgewertet werden, nicht definiert und implementierungsabhängig ist.

Was die letzte Zeile wirklich bedeutet, ist so etwas wie folgender Code, da `operator+()` nach wie vor von links nach rechts ausgeführt wird:

```
return
  operator+(
    operator+(
      operator+( string("size = "),
                 itoa(size_,buf,10) ) ,
      ", y = " ) ,
    itoa(used_,buf,10) );
```

Sagen wir, dass `size_` gleich 10 ist und `_used` gleich 5. Wenn nun der erste Parameter des äußeren `operator+()` als Erstes ausgewertet wird, so ist die Ausgabe richtigerweise »size = 10, used = 5«, da das Ergebnis des ersten Aufrufs von `itoa()` in einem temporären `string`-Objekt gespeichert wird, bevor der zweite Aufruf von `itoa()` den gleichen Buffer `buf` zum zweiten Mal verwendet. Wenn jedoch der zweite Parameter des äußeren `operator+()` als Erstes ausgewertet wird (wie das bei einigen populären Compilern der Fall ist), wird die Ausgabe fälschlicherweise »size = 10, used = 10« sein, da zunächst das äußere `itoa()` ausgeführt wird, danach überschreibt der Aufruf des inneren Aufrufs von `itoa()` das Ergebnis des ersten Aufrufs, bevor einer der beiden Werte benutzt wird.

☒ Typischer Fehler

Schreiben Sie niemals Code, der von der Reihenfolge der Auswertung von Funktionsargumenten abhängt.

```
bool Invariant()
{
  if( used_ > 0.9*size_ ) Resize( 2*size_ );
  return used_ <= size_;
}
```

Der Aufruf von `Resize()` ist mit zwei Problemen verbunden.

1. Im jetzigen Fall würde das Programm gar nicht funktionieren, da, wenn die Bedingung wahr ist, `Resize()` aufgerufen wird, wodurch sofort wieder `Invariant()` aufgerufen wird, wo die Bedingung immer noch wahr ist, wodurch wiederum `Resize()` aufgerufen wird, wodurch ... – nun, ich denke, Sie haben es begriffen.

2. Was, wenn sich der Autor von `AAssert()` aus Effizienzgründen dazu entschließt, die Fehlerausgabe zu entfernen und einfach nur »`assert( p->Invariant() );`« zu schreiben? Dann verkommt dieser Client-Code zu beklagenswertem Stil, da Code mit Seiteneffekten innerhalb eines `assert()`-Aufrufs steht. Das heißt, dass das Verhalten des Programms unterschiedlich ist, je nachdem, ob es im Debug-Modus kompiliert wird oder nicht. Selbst ohne das erste Problem wäre dies schlecht, da das bedeutet, dass das `Array`-Objekt zu unterschiedlichen Zeitpunkten in Abhängigkeit vom Kompiliermodus vergrößert wird. Das macht das Leben von Testern zur Tortur, wenn sie versuchen, Kundenprobleme mit einer Debug-Version des Programms nachzuvollziehen, das im Endeffekt ganz andere Laufzeit-Speichereigenschaften hat.

   Die Moral davon ist: Schreiben Sie keinen Code mit Seiteneffekten innerhalb eines `assert()`-Aufrufs (oder etwas, was ein solcher sein könnte) und stellen Sie immer sicher, dass eine Rekursion wirklich terminiert.

```
private:
  T*     buffer_;
  size_t used_, size_;
};

int f( int& x, int y = x ) { return x += y; }
```

   Der Default-Wert für den zweiten Parameter ist auf keinen Fall gültiges C++, so dass dieser Code nicht von einem standard-konformen Compiler übersetzt werden sollte (auch wenn einige System ihn akzeptieren). Für den Rest dieser Lektion wollen wir annehmen, dass der Default-Wert für `y` gleich 1 ist.

```
int g( int& x )            { return x /= 2; }

int main( int, char*[] )
{
  int i = 42;
  cout << "f(" << i << ") = " << f(i) << ", "
       << "g(" << i << ") = " << g(i) << endl;
```

   Hier sind wir wieder beim Problem der Auswertungsreihenfolge von Funktionsargumenten. Da es keine Möglichkeit gibt, die Reihenfolge vorherzusagen, in der `f(i)` und `g(i)` ausgeführt werden (oder auch die zwei einfachen Auswertungen von `i` selbst), könnte das ausgegebene Ergebnis völlig falsch sein. Eine Beispielausgabe

von MSVC ist »f(22) = 22, g(21) = 21«, was darauf hinweist, dass der Compiler sehr wahrscheinlich alle Funktionsargumente von rechts nach links auswertet.

Aber ist das Ergebnis nicht falsch? Nein, der Compiler hat Recht – und ein anderer Compiler könnte etwas anderes ausgeben lassen und trotzdem auch Recht haben, da der Programmierer sich auf etwas verlässt, was in C++ nicht definiert ist.

☒ Typischer Fehler

Schreiben Sie niemals Code, der von der Reihenfolge der Auswertung von Funktionsargumenten abhängt.

```
  Array<char> a(20);
  cout << a.PrintSizes() << endl;
}
```

Hierdurch sollte natürlich »size = 20, used = 0« ausgegeben werden, aber aufgrund des bereits besprochenen Fehlers in `PrintSizes()` führen einige populäre Compiler zu der Ausgabe »size = 20, used = 20«, was offensichtlich inkorrekt ist.

Vielleicht hatte Dorothy doch nicht ganz Recht mit ihrer Aussage über die Tierschau – die folgende Aussage könnte treffender sein:

»Parameter und globale Variablen und Exceptions, oh Gott!«
- Dorothy nach einem C++-Kurs

Nachwort

Wenn Ihnen die Denkaufgaben und Programmierprobleme in diesem Buch gefallen haben, habe ich gute Neuigkeiten für Sie. Dies ist nicht das Ende, da *Guru of the Week #3* nicht die letzte Ausgabe von *GotW* war und ich habe auch noch nicht aufgehört, Artikel für verschiedene Programmiermagazine zu schreiben.

Heute werden im Internet regelmäßig neue *GotW*-Ausgaben in der Newsgroup *comp.lang.c++.moderated* veröffentlicht, diskutiert und debattiert. Sie sind auf der offiziellen *GotW*-Website *www.PeerDirect.com/resources* archiviert. Als ich diese Zeilen im Juni 1999 schrieb, waren wir bereits bei #55. Um Ihnen einen Vorgeschmack für das zu geben, was noch kommt, hier ein paar Beispiele, was in den nächsten *GotW*-Ausgaben an neuem Material geboten wird:

- Mehr Informationen zu so populären Themen wie dem sicheren Gebrauch von `auto_ptr`, Namensräumen sowie Exception-Sicherheitsprobleme und -techniken, womit die Lektionen 8 bis 17, 31 bis 34 und 37 fortgesetzt werden.
- Eine dreiteilige Serie über Referenzzählung und Copy-On-Write-Techniken, einschließlich ungewöhnlicher Performance-Implikationen in multithreaded Umgebungen mit umfassendem Testcode und statistischen Messungen. Sie werden hier Material finden, das Sie gewöhnlich nirgendwo sonst finden.
- Viele Denkaufgaben über den sicheren und effektiven Einsatz der Standardbibliothek, insbesondere Container (wie `vector` und `map`) und Standard-Streams. Dies umfasst auch weiter gehende Informationen, wie die Standardbibliothek am besten erweitert wird im Sinne der Lektionen 2 und 3.
- Ein niedliches Spiel: Wie kann ein MasterMind-Spiel mit so wenig Anweisungen wie möglich geschrieben werden?

Und das sind nur einige wenige Beispiele. Wenn es genügend Interesse an dem Buch gibt, was Sie gerade in Ihren Händen halten, so ist meine Absicht eine weitere Ausgabe zu produzieren, die das nächste Bündel *GotW*-Ausgaben in weiter führender und neu organisierter Form enthält, wiederum einschließlich Texten aus anderer C++-Artikeln und Kolumnen, die ich für den »*C++-Report*«, das »*C/C++ User Journal*« und andere Magazine schreibe.

Ich hoffe, dass Ihnen das Buch gefallen hat und dass Sie mich weiterhin wissen lassen, welche interessanten Themen Sie in der Zukunft behandelt sehen möchten; werfen Sie einen Blick auf die oben aufgeführte Website, um zu sehen, wie Sie mir solche Anfragen zukommen lassen können. Einige der Themen, die Sie in diesem Buch finden, wurden durch solche Mails veranlasst.

Nochmals Dank an all die, die ihr Interesse und ihre Unterstützung für *GotW* und dieses Buch ausgedrückt haben. Ich hoffe, Ihnen hilft dieses Material bei Ihrer täglichen Arbeit, wenn Sie schnellere, sauberere und sicherere C++-Programme schreiben.

Bibliographie

Barton94: John Barton und Lee Nackman, *Scientific and Engineering C++.* Addison-Wesley, 1994.

Cargill92: Tom Cargill, *C++ Programming Style.* Addison-Wesley, 1992.

Cargill94: Tom Cargill, »Exception Handling: A False Sense of Security.« *C++ Report*, 9(6), Nov.-Dec. 1994.

Cline95: Marshall Cline und Greg Lomow, *C++ FAQs.* Addison-Wesley, 1995.

Cline99: Marshall Cline, Greg Lomow und Mike Girou, *C++ FAQs, Second Edition.* Addison-Wesley Longman, 1999.

Coplien92: James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

Ellis90: Margaret Ellis und Bjarne Stroustrup, *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

Gamma95: Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Keffer95: Thomas Keffer, *Rogue Wave C++ Design, Implementation, and Style Guide.* Rogue Wave Software, 1995.

Koenig97: Andrew Koenig und Barbara Moo, *Ruminations on C++*. Addison-Wesley Longman, 1997.

Lakos96: John Lakos, *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

Lippman97: Stan Lippman (ed.), *C++ Gems.* SIGS / Cambridge University Press, 1997.

Lippman98: Stan Lippman und Josée Lajoie, *C++ Primer, Third Edition.* Addison-Wesley Longman, 1998.

Martin95: Robert C. Martin, *Designing Object-Oriented Applications Using the Booch Method*. Prentice-Hall, 1995.

Martin00: Robert C. Martin (ed.), *C++ Gems II*. SIGS / Cambridge University Press, 2000.

Meyers97: Nathan Meyers, »The Empty Base C++ Optimization.« *Dr. Dobb's Journal,* Aug. 1997.

Meyers96: Scott Meyers, *More Effective C++*. Addison-Wesley, 1996.

Meyers98: Scott Meyers, *Effective C++, Second Edition.* Addison-Wesley Longman, 1998.

Meyers99: Scott Meyers, *Effective C++ CD.* Addison-Wesley Longman, 1999. (See also http://meyerscd.awl.com)

Murray93: Robert Murray, *C++ Strategies and Tactics*. Addison-Wesley, 1993.

Stroustrup94: Bjarne Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1994.

Stroustrup97: Bjarne Stroustrup, *The C++ Progamming Language, Third Edition.* Addison-Wesley Longman, 1997.

Sutter98: Herb Sutter, »C++ State of the Union.« *C++ Report,* 10(1), Jan. 1998.

Sutter98(a): Herb Sutter, »Uses and Abuses of Inheritance, Part 1.« *C++ Report,* 10(9), Oct. 1998.

Sutter99: Herb Sutter, »Uses and Abuses of Inheritance, Part 2.« *C++ Report,* 11(1), Jan. 1999.

Index