3-8273-1711-8
Exceptional C++

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 exception-sicher 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.

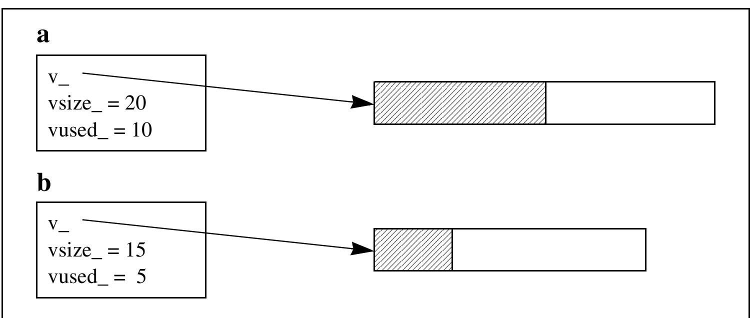

Abbildung 3.1: Zwei `StackImpl<T>`-Objekte a und b

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

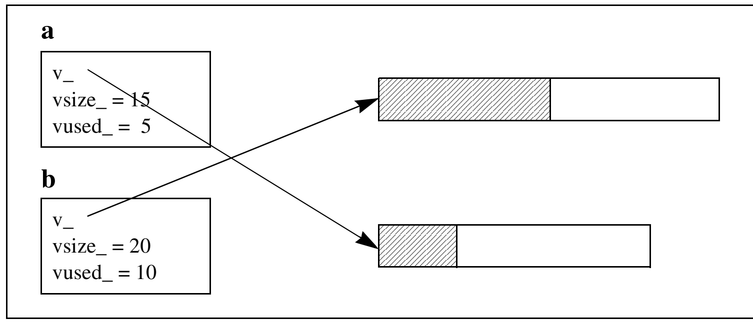

Abbildung 3.2: Die gleichen `StackImpl<T>`-Objekte a und b nach `a.Swap(b)`

`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.

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.