Kapitel 5
Ausnahmen und Zeigerverwaltung

Ausnahmen und Zeigervariablen scheinen auf den ersten Blick nur wenig miteinander zu tun zu haben. Außerdem haben wir uns ja bereits in einem der vorhergehenden Kapitel bereits ausführlich mit Zeigervariablen beschäftigt. Wenn die Anwendungen komplizierter werden und mit Vererbungshierarchien und polymorphen Funktionen gearbeitet werden muss, stellt man allerdings schnell fest, dass die aufgestellten Regeln für die Erzeugung von Zeigerobjekten und deren Vernichtung leider nicht so einfach und eindeutig handhabbar sind. Objekte werden an verschiedene Programmbereiche weitergereicht, die Kopien erstellen oder nur den Zeigerwert auf einer eigenen Variablen festhalten, und nach einigem Hin und Her kann man die Übersicht verlieren, ob (*zur Laufzeit*) ein Objekt noch existiert und vernichtet werden muss oder bereits der nächste Funktionsaufruf im Niemandsland endet. Besonders auffallend ist dies bei Ausnahmen, die formal betrachtet nichts anderes als `GOTO`-Anweisungen über Funktionsgrenzen hinaus darstellen, so dass sich die gemeinsame Untersuchung von Verwaltungsstrategien geradezu aufdrängt. Auch die Abwicklungsverfahren von Ausnahmen sind natürlich für sich gesehen ebenfalls schon sehr interessant.

An dieser Stelle ein Wort zur Konkurrenz: Java-Entwickler nehmen gerne für sich in Anspruch, wesentlich sicherer programmieren zu können als C++-Entwickler, da keine behandlungsfehleranfälligen Zeigerkonstrukte in Java auftreten und grundsätzlich mit Ausnahmen gearbeitet wird. Die Behandlungsfehleranfälligkeit haben wir mit unseren einfachen Regeln für die Zeigerbehandlung bereits erheblich entschärft, und das unbedingte Werfen von Ausnahmen setzt voraus, dass diese auch an der richtigen Stelle gefangen werden, also eine Übersicht des Programmierers über den gesamten Programmbereich. Wir werden mit den Werkzeuge, die wir im weiteren Verlauf dieses Kapitels entwickeln werden, diesen Vorurteilen locker begegnen können:

- Wir werden strategische Klassen entwickeln, die Zeiger automatisch – auch beim Wurf von Ausnahmen – verwalten und je nach Intention des Programmierers bei Zuweisung an andere Variable Kopien der ursprünglichen Variable erstellen oder nur den Zeigerwert übernehmen (*Java kennt nur das zweite Prinzip, die Übergabe von Referenzen. Eine Kopierabsicht muss manuell vom Programmierer*

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_5, © Springer-Verlag Berlin Heidelberg 2010

umgesetzt werden). Dafür ist nur eine entsprechende Deklaration der Variablen notwendig.

- Wir werden eine Verwaltung entwickeln, die laufzeitabhängig entscheidet, ob eine anstehende Ausnahme geworfen oder das Programm fortgesetzt wird (*in diesem Fall kommen meist logische Aussagen zum Einsatz, um den weiteren Ablauf zu steuern*). Dazu muss im Programm durch Deklaration spezieller Variablen nur vorgegeben werden, ob eine Ausnahme in den aufgerufenen Funktionen erwünscht ist oder nicht (*Java verfügt nicht über Mechanismen, das Ausnahmemanagement nach Bedarf an oder abzuschalten*).

Insgesamt schaffen wir uns damit in C++ die Möglichkeit, verschiedene Objektbehandlungsstrategien – Funktionen mit oder ohne polymorphes Verhalten, Objektkopien oder Referenzen – und Ausnahmebehandlungsstrategien überwiegend durch simple Variablendeklaration objekt- und funktionsgenau einstellen zu können, ohne während der Erzeugung des Kodes noch auf etwas achten zu müssen. Da in Java außer der Hauptstrategie nichts per Deklaration zur Verfügung steht, sondern durch entsprechenden Code realisiert werden muss, darf man das Sicherheitsargument wohl getrost „ad acta“ legen.

Prinzipiell ist zu Ausnahmen anzumerken, dass ihre Verwendung mit einem beträchtlichen internen Aufwand versehen ist. Der C++ Compiler erlaubt deshalb mit gutem Grund, Ausnahmen zu deaktivieren. Ausnahmen sollten deshalb auch nur dort Verwendung finden, wo die größere Laufzeit keine Probleme verursacht. Bei der Programmierung von Methoden sollte man andererseits nicht davon ausgehen, dass Ausnahmen aktiviert sind, und bei Ausnahmesituationen neben dem Werfen der Ausnahme auch eine Signalisierung des Problems durch spezielle Rückgabewerte vorsehen.

5.1 Zur Arbeitsweise mit Ausnahmen

Ausnahmen oder „Exceptions“ sind eine Möglichkeit, auf Abweichungen vom normalen Ablauf eines Programms zu reagieren. Leider wird der Begriff „Ausnahme“ häufig als eine Laufzeithilfe missverstanden, auf Programmierfehler zu reagieren.[1]

[1] In fast allen Dokumenten findet sich die Beschreibung: „Ausnahmen dienen zur Behandlung nicht vorhergesehener Programmfehler“, gefolgt von dem Beispiel, eine Quadratwurzel aus einer negativen Zahl zu ziehen. Darin stecken eine ganze Menge von Ungereimtheiten: (a) Wenn der Fehler nicht vorherzusehen war, wie kommt der Programmierer dann auf die Idee, eine Ausnahmebehandlung dafür zu programmieren? (b) Wenn jemand einen Algorithmus programmiert, in dem Wurzeln aus negativen Zahlen gezogen werden können, aber keine komplexen Zahlen für deren Aufnahme bereitstehen, hat derjenige dann nicht eher ein mathematisches als ein programiertechnisches Problem? (c) Wenn ein solcher Fehler auftritt, ist nichts zu reparieren, sondern das Programm ist zu beenden. Dann hätte aber auch eine Ausgabe auf „cerr“ genügt, gefolgt von einem „exit()“, und man hätte sich den Aufwand mit dem Ausnahmemanagement sparen können.

Zugegeben, Ausnahmen eignen sich natürlich auch dafür, Programme nach Auftreten eines Fehlers (*sehr selten*) wieder auf Kurs beziehungsweise sie zumindest (*meistens*) sauber zu einem Ende zu bringen, und da sich „*es hat sich eine Ausnahme ereignet* “ deutlich freundlicher anhört als „*aufgrund eines Programmierfehlers muss das Programm abgebrochen werden* “, wird der Begriff „Ausnahme“ auch an Stellen benutzt, an denen er eigentlich wenig verloren hat. Halten wir jedoch für uns folgenden Hauptnutzungsrahmen fest.

> **Definition.** Eine Ausnahme ist eine selten auftretende, aber völlig normale Situation in einem Programmablauf, die im Programmdesign berücksichtigt und beschrieben ist. Die Einleitung einer Ausnahmebehandlung ist keine Reaktion auf einen nicht beabsichtigten Fehler, sondern eine bewusst programmierte Anweisung zur einfachen und korrekten Behandlung einer besonderen Situation.[2]

Ich referiere hier nur kurz den formalen Einsatz von Ausnahmen als Wiederholung der Grundkenntnisse. Eine Anweisungssequenz mit Ausnahmebehandlung besteht aus einem Programmblock, der den „normalen“ Programmablauf beschreibt, sowie einem oder mehreren Blöcken für die Bearbeitung des Ungewöhnlichen

```
try {
    ...             // normaler Programmablauf
} catch (A a){
    ...             // Ablauf bei Ausnahme  „ A"
} catch (B * b) {
    ...             // Ablauf bei Ausnahme  „ B"
    delete b;       // !! siehe Beschreibung !!
} catch (C& c) {
    ...             // Ablauf bei Ausnahme  „ C"
} catch(...){
    ...             // Fangen, was noch übrig bleibt
}//end try
```

Das Bild ähnelt einer Liste polymorpher Funktionen, die sich durch unterschiedliche Übergabeparameter unterscheiden. In den Wurfanweisungen für eine Ausnahme können Parameter übergeben werden, die Informationen über die Ursache der Ausnahme enthalten und Entscheidungen über die weitere Vorgehensweise nach dem Fangen erlauben. Je nach Typ der geworfenen Information wird die passende `catch`-„Funktion“ angesprungen und ausgeführt (*im Weiteren werden wir von* `catch`*-Blöcken sprechen, da es sich hier trotz des Aussehens nicht um Funktionsmechanismen handelt, wie wir noch sehen werden*). Es sind alle Parametertypen möglich, die auch in normalen Methoden auftreten, allerdings ist die Anzahl auf

---

[2] Darauf weist eigentlich auch schon der Name hin: „exception“ bedeutet „Ausnahme“ und nicht „Fehler“. Trotzdem wird es hierzulande meist mit „Fehlermanagement“ übersetzt, als ob die Amerikaner ihrer eigenen Sprache nicht mächtig wären und den Begriff „error management“ mit „exception management“ verwechselt hätten.

einen Parameter beschränkt, d.h. bei größeren zu übertragenden Datenmengen ist ein geeigneter `struct`-Typ zu erzeugen, der dazu in der Lage ist.

Die Ausnahme selbst wird produziert durch Anweisungen der Art (*Parametererzeugung in der Reihenfolge der Nutzung in den Ausnahmeblöcken*)

```
...
throw A();
...
throw new B();
...
throw C();
...
```

Diese können sich an beliebiger Stelle im Kontrollbereich des `try`-Blockes befinden, also direkt innerhalb des Blockes oder in dort aufgerufenen Methoden. Damit ist auch klar, dass es sich bei `catch`-Blöcken nicht um Funktionen handeln kann und der gesamte Ausnahmemechanismus dynamisch bearbeitet wird. Betrachten Sie das Beispiel

```
void f1(){ throw "function1";}
void f2(){
   try{ f1(); }
   catch(...){ throw int(5); }
}//end function
void (*ff)();
int main(){
    try{
        ...
        if(i==0)
            ff=&f1;
        else
            ff=&f2;
        ff();
        cout << "good" << endl;
    }catch(char const* s){
        cout << s << endl;
    }catch(int i){
        cout << i << endl;
    }//endcatch
```

Beim Fangen der Ausnahmen müssen Aufrufstacks rückgebaut werden, statt wie bei Funktionsaufrufen aufgebaut zu werden. Außerdem wird die in `f1()` geworfene Ausnahme je nach Programmablauf in der Funktion **f2()** oder in `main()` gefangen. Das lässt sich nur noch sinnvoll auflösen, wenn vor beziehungsweise während des Rückbaus der Aufrufstacks dynamisch geprüft wird, ob ein geeigneter `catch`-Block vorhanden ist. Wird dabei in einer Liste verschiedener Alternativen

keine Übereinstimmung gefunden, so wird der komplette `catch`–Block übersprungen und die Suche in der diesen Programmteil rufendenden Methode fortgesetzt. Das folgende Programm überspringt beispielsweise den `catch`–Block für den Parameter des Typs „**B**“ und führt bei einer Ausnahme den Block „**A**“ aus:

```
try {
    ...
    try{
        ...
        throw A();
        ...
    }catch(B b){
        ...         // Block  „ B"
    }//end try
    ...
}catch(A a){
    ...             // Block  „ A"
}//end try
```

Ist ein passender `catch`-Block gefunden, so wird nur der darin enthaltenen Code ausgeführt und eventuell vorhandene weitere `catch`–Blöcke werden übersprungen. Beim Einfangen der Ausnahme werden Vererbungsbeziehungen ebenfalls in gewohnter Weise unterstützt. Die Vererbung `class A: public B` führt in der oben angegebenen Beispielsequenz dazu, dass doch Block `B` die Ausnahme fängt und nur durch eigenes Weiterwerfen die Meldung an den Block `A` weiterleitet.

Schachtelungen von `try-catch`-Blöcken und `throw`–Anweisungen in `catch`–Blöcken sind, wie die Beispiele zeigen, ebenfalls erlaubt. Die einzige Ausnahme bilden Destruktoren, in denen keine Ausnahmen geworfen werden dürfen (*warum, diskutierten wir später*). Allerdings können `throw`–Anweisungen in `catch`–Blöcken nicht dazu genutzt werden, die Kontrolle an einen anderen `catch`–Block im gleichen `try-catch`-Gesamtblock weiterzureichen, beispielsweise von `catch(A a)` an `catch(B* b)` im ersten Beispiel. Dies funktioniert nur in geschachtelten Ausnahmeblöcken, indem beispielsweise in `catch(B b)` im letzten Beispiel eine ausnahme geworfen wird, die bei `catch(A a)` im übergeordneten Block gefangen wird.

Nun weiß man häufig nicht, welche Ausnahmen geworfen werden, wenn der Funktionskode nicht daraufhin untersucht wird. Es ist deshalb im Sprachstandard vorgesehen, dass eine Funktion in der Definition bekannt machen kann, welche Ausnahmen geworfen werden:

```
void f() throw() ;  // wirft keine Ausnahmen
void f() throw(char*, int);
                    // wirft Ausnahmen der Typen
                    // char* oder int
```

Der Programmierer, der diese Funktionen nutzt, muss daher nur Vorkehrungen für Ausnahmen der beschriebenen Art treffen. Allerdings ist das mit etwas Vorsicht zu genießen:

```
void f() throw(char const*){
    ...
    throw int(5);
    ...
}//end function
int main(){
    ...
    try{
       f();
    }catch(...){ ...
```

Die Funktion `f()` ist so deklariert, dass nur Ausnahmen des Typs `char const*` von ihr geworfen werden sollen, im Inneren aber eine Ausnahme des Typs `int` erzeugt wird (*das muss nicht so evident falsch wie hier sein, sondern kann ja auch in einer aufgerufenen weiteren Funktion geschehen*). Es wird gemäß Sprach-Standard nun aber weder das Werfen von `int` verhindert noch die Ausnahme an dem vom Programmierer vorsorglich eingerichteten Punkt `catch(...)` gefangen, sondern die Anwendung rennt ungebremst auf die allgemeine Bedingung „unerwarteter Zustand“ und wird mit diesem Kommentar beendet (*was auch passiert, wenn eine Ausnahme mit einem Datentyp geworfen wird, für den kein fangender* `catch`*-Block implementiert wurde*). Die `throw`-Definition im Funktionskopf darf also nur dann verwendet werden, wenn der Programmierer sicher ist, dass wirklich nichts anderes erzeugt wird, er also andere Ausnahmen aus aufgerufenen Funktionen selbst fängt. Wenn er dazu keine Lust hat, sollte er tunlichst auf dieses Sprachelement verzichten.

Aus dieser Arbeitslogik wird nochmals deutlich, dass sich die Ausnahmebehandlung nicht auf unvorhergesehene Fehler beschränkt. Für die Ausgabe eines Textes wie „*sie versuchen gerade, durch Null zu dividieren* “ ist das ganze nämlich viel zu kompliziert. Die Verwendung anderer Übergabeparameter als einfacher Texte für solche Einsatzfälle ist häufig noch fataler, als gar nichts zu tun, denn bei Verwendung von Ausnahmen muss genau darauf geachtet werden, dass auch ein `catch`–Block in der Anwendung vorhanden ist, der in der Lage ist, eine geworfene Ausnahme zu fangen.

Formal sieht der `try-catch`–Formalismus also so ähnlich aus wie ein normaler Funktionsaufruf – allerdings rückwärts, aus dem Stack heraus. Der Vorgang ist irreversibel: ein Rücksprung nach der Fehlerbehandlung an die Position nach einer `throw`–Anweisung ist nicht möglich,[3] und das hat natürlich eine Reihe von Konsequenzen. Beispielsweise muss bei der Verwendung von Zeigervariablen in `throw`–Anweisungen die Variable anschließend freigegeben werden, wie das erste

[3] Eigentlich sollte man gerade so etwas als eine der Optionen bei „unvorhergesehenen“ Fehlern erwarten: Richtigstellen der fehlerhaften Daten und Fortsetzen der Operation.

Beispiel oben zeigt. Durch das rückwärts–Aufrollen des Stacks ist auch die Variablenverwaltung im Ausnahmemanagement alles andere als trivial: sämtliche im `try`-Block deklarierte Variablen sowie auch alle in aufgerufenen Funktionen oder in Unterblöcken deklarierte Variablen werden automatisch freigegeben. Lediglich die in der `throw`–Anweisung geworfene Variable ist im `catch`–Block noch gültig.[4] Die in den freigegebenen Variablen vorhandene Information ist bei Eintritt in den `catch`–Block natürlich auch verschwunden, wenn sie nicht zuvor irgendwo anders gesichert wurde (*auch das passt nicht zum Bild des „Unvorhergesehenen“*). Außerdem stoßen wir hier auch auf ein anderes ernstes Problem. Sehen wir uns dazu das folgende Programmfragment an

```
int function(){
    A*a=new A();
    ...
    throw "Ausnahme eingetreten"
    ...
    delete a;
    return 1;
}//end function
```

Die Zeigervariable `a` wird hier im Falle einer Ausnahme nicht mehr freigegeben, weil bei einem `throw` die delete–Anweisung nicht mehr erreicht wird. Da der Platzhalter verschwunden ist, besteht aber auch keine Möglichkeit, auf sie noch einmal zuzugreifen. Ähnlich problematisch ist eine Zeigerübergabe an die rufenden Programmteile als Rückgabewert einer Funktion, da die Eigentumskette unterbrochen wird und gegebenenfalls (*neben der verlorenen Zeigervariablen*) jemand versucht, eine Variable freizugeben, die nicht existiert (*verifizieren Sie, dass zumindest dies bei strikter Einhaltung unserer Initialisierungsregeln für Zeigervariable nicht auftreten kann*). Wir untersuchen daher im weiteren Techniken, die eine Arbeit mit Zeigervariablen in einer Umgebung mit Ausnahmemanagement zulässt.

Bemerkungen. Das Ausnahmemanagement ist aufwändig und kann den Programmablauf stark verlangsamen, da ja bei jedem Funktionsaufruf Vorkehrungen für den Ausnahme-Stack-Rückbau zu treffen sind. C++-Compiler erlauben daher die Aktivierung oder Deaktivierung des Ausnahmemanagements je nach Bedarf.

---

[4] Wird sie nicht, wie in den Beispielen, erst in der „throw“-Anweisung erzeugt, sondern ist zuvor schon deklariert (A a; ... throw a; ..), so wird mittels des Kopierkonstruktors eine Kopie für den „catch“-Block erzeugt und die ursprüngliche Variable freigegeben. Ist aber gar kein Kopierkonstruktor definiert und auch nicht gesperrt, verwenden viele Entwicklungsumgebungen an dieser Stelle selbst implementierte „default“-Konstruktoren. Je nach Objekt können die Folgen übelster Natur sein.

5.2 Typermittlung und Zugriffsstandardisierung

5.2.1 Ableitung definierter Typen

Wir geben hier zunächst eine allgemeine Methode an, wie die expliziten Typen aus einem beliebigen Template-Parametertyp generiert werden können. Techniken dieser Art werden bereits in den Container- und Iteratorklassen verwendet, werden jedoch erst hier für unsere Arbeit interessant, so dass wir sie anwendungsnah detaillierter diskutieren.

Bei einer Unterscheidung von Typen kann man überschneidend zwei Kategorien untersuchen:

(a) Die Typart wie fundamentale Typen, Felder, Klassen, Funktionen oder klassengebundene Methoden,
(b) die Übergabeart wie Wertparameter, Zeiger, Referenzen sowie die konstanten Versionen davon.

Das Ergebnis der Klassifizierung sind einerseits `enum`-Konstante, die weitere Weichenstellungen in Compileralgorithmen mittels Templatetechniken erlauben, andererseits können auch streng definierte Typen für Variablen und Übergabeparameter bereitgestellt werden. Wir beginnen hier mit dem zweiten Ziel, das sich vorzugsweise auf Kategorie (b) beschränkt. In der einer beliebigen Templateklasse, beispielsweise einer Iteratorklasse, spezifizieren wir die Typen mittels eines Hilfstemplates.

```
template <class T> class A {
public:
   typedef typename
      TypeCheck<T>::ValueType      value_type;
   typedef typename
      TypeCheck<T>::PointerType    pointer;
   typedef typename
      TypeCheck<T>::ReferenceType  reference;
   ...
```

Diese hat die Aufgabe, bei beliebigen Instanziierungen wie

```
A<int> a;
A <double*> b;
A <string#> c;
```

in Anwendungsteilen wie

```
template <class T> void func(A<T>& a){
   typename A<T>::value_type r;
   ...
```

für eine saubere Typdefinition zu sorgen.

Aufgabe. Ist zwar alles schon mehrfach erläutert worden, aber nur zur Wiederholung: welche Aufgabe hat in diesen Ausdrücken die Spezifizierung `typename`, warum muss sie angegeben werden?

`TypeCheck<..>` selbst löst nun das Problem, den als Template-Parameter übergebenen Typ, Zeigertyp oder Referenztyp als solchen zu identifizieren und sauber auf Grundtypen abzubilden, und zwar mit Hilfe einer internen Templateklasse und mehreren Spezialisierungen:

```
template <class T, class A=AccessPolicy<T> >
class TypeCheck {
private:
    template <class U> struct TypeChecker{
        enum { result=0 };
        typedef U* PointerType;
        typedef U  ValueType;
        typedef U& ReferenceType;
    };//end struct

    template <class U> struct TypeChecker<U*>{
        enum { result=1 };
        typedef U* PointerType;
        typedef U  ValueType;
        typedef U& ReferenceType;
    };//end struct

    template <class U> struct TypeChecker<U&>{
        enum { result=2 };
        typedef U* PointerType;
        typedef U  ValueType;
        typedef U& ReferenceType;
    };//end struct

public:
    enum { isPtr = TypeChecker<T>::result==0,
           isVal = TypeChecker<T>::result==1,
           isRef = TypeChecker<T>::result==2};
    typedef A Policy;
    typedef T                          BaseType;
    typedef typename
      TypeChecker<T>::PointerType      PointerType;
    typedef typename
      TypeChecker<T>::ValueType        ValueType;
    typedef typename
      TypeChecker<T>::ReferenceType    ReferenceType;
};//end class
```

Die Klasse gibt zusätzlich über eine Konstante an, welchen Typ der ursprüngliche Templateparameter aufweist, sowie über `BaseType` eine Typisierung in der Art des Templateparameters. Im Vorgriff haben wir auch bereits eine Hilfsklasse eingebaut, die einen einheitlichen Zugriff auf Objekte ermöglicht und die wir im nächsten Teilabschnitt vorstellen.

Aufgabe. Wie können die `const`-Versionen der Typen in den Typtester eingebaut werden? Ist es sinnvoll, auch Zahlenparameter zu testen?

5.2.2 Zugriffsnormierung

Die Typauflösung erlaubt nun die Deklaration definierter Typen unabhängig von der Übergabeart des Templateparameters, für den Zugriff auf Attribute in der Templateklasse selbst ist die Hilfsklasse `AccessPolicy` schon präventiv vorgesehen wird. Diese wird ebenfalls durch mehrere Spezialisierungen realisiert, die eine einheitliche Zugriffsschnittstelle liefern:

```
template <class T> struct AccessPolicy {
  inline T*       ptr(T& t)      const {return &t;}
  inline T const* ptr(T const& t)const {return &t;}
  inline T&       ref(T& t)      const {return t;}
  inline T const& ref(T const& t)const {return t;}
  inline void construct(T&){}
  inline void destruct(T&){}
};//end struct

template <class T> struct AccessPolicy<T*> {
  inline T*       ptr(T* t)      const{return t;}
  inline T const* ptr(T const* t)const{return t;}
  inline T&       ref(T* t)      const{return *t;}
  inline T const& ref(T const* t)const{return *t;}
  inline void construct(T*& t){t=new T();}
  inline void destruct(T*& t) {delete t;}
};//end struct
...
```

Beachten Sie, dass bei dieser Gelegenheit gleich das Problem der Instanziierung eines Attributs mitgelöst ist. `construct` und `destruct` sind in der Lage, auch Zeigerattribute korrekt zu Instanziieren und zu Beseitigen:

```
template <class T> class A {
public:
   A(){ TypeCheck<T>::Policy::construct(obj); }
   virtual ~A(){ TypeCheck<T>::Policy::destruct(obj);}
private:
   T obj;
};
```

5.2.3 Ermittlung der Typart

Relativ simpel ist die Unterscheidung zwischen Fundamentaltypen und selbstdefinierten Typen, die über eine Reihe von Spezialisierungen durchführbar ist:

```
template <class T> struct isFundamental {
   enum { yes = false };
};

template <> struct isFundamental<char> {
   enum { yes = true };
};

...
```

In dieser Form können Typen überprüft werden. Alternativ lässt sich auch durch eine Templatefunktion die Überprüfung an einer Variablen durchführen:

```
template <class T> bool checkFundamental(T){
   return isFundamental<T>::yes;
}
```

Die Funktionsform kann auch eingesetzt werden, wenn Variablen nicht vorhanden sind. Das dabei auftretende Problem, eine temporäre Variable des Typs `T` deklarieren zu müssen, was u.U. nicht zulässig ist, lässt sich mit der weiter vorne vorgestellten `type2type`-Technik umgehen.

> **Aufgabe.** Spezialisieren Sie die Methode `checkFundamental` für `type2type`.

Will man im Falle eines Fundamentaltypen genauere Informationen erhalten, lassen sich in gleicher Art Tests wie

```
template <class T> struct isChar { ...
```

definieren.

Die Prüfungen funktionieren allerdings nur mit Werttypen, nicht aber mit Zeigern oder Referenzen. Mit dem `TypeCheck` lässt sich aber auch dieses Problem umgehen, indem die Prüfmethode folgendermaßen umgeschrieben wird:

```
template <class T> bool checkFundamental(T){
   return isFundamental<typename
                        TypeCheck<T>::value_type>::yes;
}
```

Wenn ein Zeigertyp vorliegt, lässt sich über das dahinter verborgene Objekt meist wenig mehr ermitteln, wie etwa die Frage, ob es sich um ein einzelnes Objekt handelt oder um ein Feld. Bei direkt als Feld deklarierten Typen sieht die Situation besser aus:

```
template <class T> struct isField {
   enum { yes=false, size=0 };
};

template <class T, int N> struct isField<T[N]> {
   enum { yes=true, size=N };
};

template <class T> struct isField<T[]> {
   enum { yes=true, size=-1 };
};
```

Aufgabe. Auch hier lassen sich wieder Templatefunktionen für den einfacheren Zugriff definieren sowie Rückgriffe auf `type2type` und `TypeCheck` zu Berücksichtigung spezieller Randbedingungen. Dies sei Ihnen überlassen.

Zeiger auf Klassenattribute lassen sich allerdings als solche identifizieren:

```
template <class T> isMemberPtr{
   enum { yes=false };
};

template <class T, class C> isMemberPtr<T C::*> {
   enum { yes=true };
   typedef T MemberType;
   typedef C ClassType;
};
```

Bei Templateparametern muss es sich aber nicht unbedingt um Datentypen handeln, es können sich auch Funktionen dahinter verbergen. Diese sowie die Typen der Übergabeparameter lassen sich durch Spezialisierungen ermitteln:

```
template <class T> struct isFunction {
   enum { yes = false, parms=0 };
};

template <class R> struct isFunction<R()> {
   enum { yes = true, parms=0 };
   typedef R Func();
};

template <class R, class P1> struct isFunction<R(P1)> {
   enum { yes = true, parms=1 };
   typedef R Func();
   typedef P1 Par1;
};
...

template <class R, class P1> struct isFunction<R(P1,...)> {
   enum { yes = true, parms=-1 };
```

```
   typedef R Func();
   typedef P1 Par1;
};
```

Dabei sind so viele Spezialisierungen notwendig, wie an Ermittlung von Übergabeparametern erwünscht ist. Die Templates reagieren auf Typdefinitionen der Art

```
typedef int func(double);
...
... isFunction<func>::yes ...
```

Soll die Überprüfung direkt an Methoden durchgeführt werden, so wird im Funktionstemplate

```
void MyFunc(double){...}

template <typename F> bool checkFunction(F){ ...
...
... checkFunction(MyFunc) ...
```

allerdings ein Funktionszeiger übergeben, der nur zur korrekten Auswertung führt, wenn dieser in der Methode durch `TypeCheck` auf den Basistyp zurückgeführt wird[5]

```
template <typename F> bool checkFunction(F){
   return isFunction<TypeCheck<F>::value_type>::yes; }
```

Je nach Anwendungsfall kann man mit diesen Typuntersuchungen Oberklassen zusammenfügen, die verschiedene Eigenschaften und Typen vereinigen. Dabei sollten Sie noch klären:

> **Aufgabe.** Wie kann festgestellt werden, ob ein Typ eine Klasse ist?

Etwas unglücklich mag dabei die Arbeit mit Funktionen sein, wenn die Parameteranzahlen variieren. Um nicht Gefahr zu laufen, auf undefinierte Typen zu stoßen, kann auch ein Ersatztyp herhalten:

```
struct NoType {}

template <class R> struct isFunction<R()> {
   enum { yes = true, parms=0 };
   typedef R Func();
   typedef NoType P1;
   typedef NoType P2;
   ...
};
```

[5] Funktionszeigersymbolik in der Spezialisierung lässt der Compiler nicht zu, weshalb dieser Umweg notwendig ist.

Später im Buch werden wir dieses Problem durch die Einführung von Typlisten variabler Länge vereinfachen.

Wie Sie bei den Übungen bemerkt haben werden, ist die Programmierung solcher Analysestrukturen recht aufwändig, während die Anzahl der Anwendungen, in denen man diese Techniken tatsächlich benötigt, vermutlich überschaubar bleiben. Glücklicherweise besteht auch hier nicht die Notwendigkeit, über die Übungen hinaus größere Kodeteile zu produzieren. Die boost++ Bibliothek stellt eine größere Anzahl dieser Werkzeuge zur Verfügung, auf die man bei Bedarf zurückgreifen kann.

5.3 Verwaltung von Zeigervariablen

Bevor wir uns hier eine Reihe von Werkzeugen für die Verwaltung von Zeigervariablen erzeugen, sei kurz noch einmal reflektiert, warum wir uns damit überhaupt beschäftigen müssen. Geht es nicht auch ohne Zeiger?

Als Antwort sei Ihnen das Kapitel über Vererbung und virtuelle Methoden ins Gedächtnis gerufen. Wenn wir komplexe Umgebungen modellieren, kommen wir um Vererbung nicht herum, und an den Übergabeschnittstellen von Methoden werden nicht die Spezialisierungen übergeben, sondern die schlichteren Basisklassen typisiert. Eine Basisklasse nennen und eine Spezialisierung meinen lässt sich aber nur mit Hilfe von Zeigern realisieren. Man kommt um sie somit nicht herum.

Möglicherweise werden Sie sich erinnern, dass wir dieses Argument im Zusammenhang mit Containern auch schon einmal verwendet haben und auf das Problem gestoßen sind, wie Zeigervariable in einem Container verwaltet und sicher wieder zerstört werden können. Die Option, spezielle Allokatorklassen zu konstruieren, schien aus verschiedenen Gründen nur bedingt verlockend. Dieses Problem werden wir mit den hier zu schaffenden Werkzeugen nun auch erledigen. Es geht also nicht nur um Zeiger und Ausnahmen.

5.3.1 Manuelle Ausnahmeverwaltung

Um nun (*nicht nur*) im Falle von Ausnahmen die vorhandenen Zeiger korrekt behandeln zu können, sind eine Reihe von Strategien möglich, die mit unterschiedlichem Aufwand verbunden sind. Die einfachste Möglichkeit, die aber wieder sehr viel Aufmerksamkeit beim Programmierer erfordert, ist die Nutzung einer weiteren Eigenschaft des Ausnahmesystems, das „Weiterwerfen“ einer gefangenen Ausnahme, deren Typ unbekannt ist:

```
T* t=0;
try{
    ...
    t=new T();
    ...
}catch(...){
```

```
    if(t) delete t;
    throw;
}//endtry
if(t) delete t;
```

Der Befehl `throw` ohne Parameter sorgt dafür, dass die gefangene Ausnahme in der gleichen Form noch einmal geworfen wird, in der sie ursprünglich ausgelöst worden ist, d.h. die übergeordneten `try..catch`-Anweisungen können typgenau auf die Ausnahme reagieren.

Diese Form der Problembehandlung erfordert eine Verdopplung der Aufräumaktivitäten, was natürlich wieder fehleranfällig ist, ist aber mit unseren C-Zeigerverwaltungsregeln sicher durchzuführen.[6]

5.3.2 Platzhalter- oderTrägervariable

Da für normal deklarierte Variable das Problem, bei der Verlassen eines Bereiches nicht sauber „entsorgt“ zu werden, nicht besteht, kann die Verantwortung für ein Zeigerobjekt auch einer solchen Variablen übergeben werden:

```
template <typename T> class ptr{
public:
    T* t;
    ptr() {t=0;}
    ~ptr(){ if(t) delete t;}
};//end class
```

Diese Primitivform ist natürlich zweckmäßigerweise mit einiger Funktionalität auszustatten. Die Verwendung einer solchen Trägervariable sollte zunächst transparent sein, d.h. im weiteren Programm sollte nicht darauf geachtet werden müssen, ob eine Trägervariable oder eine Zeigervariable verwendet wird:

```
A* a     = new A();
ptr<A> b = new A();
...
a->foo();        // Funktionsaurufe gleich
b->foo();
*a = *b;      // Variablenzugriff gleich
...
```

Andererseits ist bei einer Trägervariablen im Gegensatz zu einer Zeigervariablen klar, dass es sich bei dem Attribut immer um eine Zeigervariable mit eigenem

---

[6] Java bietet an dieser Stelle das Sprachkonstrukt „finally“, das Code enthält, der nach dem try- und sämtlichen catch-Blöcken durchgeführt wird und für solche Aufräumoperationen vorgesehen ist. Da sich der finally-Code aber nicht auf das Geschehen außerhalb des try-catch-Blockes bezieht, also u.U. nicht alle aufzuräumenden Teile umfasst, und in C++ effektivere Methoden des Aufräumens existieren, hat man auf dieses Sprachkonstrukt verzichtet.

Speicherplatz handelt. Bestimmte Operationen wie Kopierkonstruktoren oder Zuweisungen zwischen zwei Trägervariablen müssen wir daher unterdrücken, um nicht zwangsläufig Fehler zu implementieren. Wir erhalten für die `ptr`-Klasse damit folgende Konstruktion:

```
template <class T> class ptr {
public:
    ptr():t(0) {}
    ptr(T* p)  {t=(T*)p;}
    ~ptr(void) {if(t) delete t; }

    ptr& operator=(T* p){
        if(t) delete t;
        t=(T*)p;
        return *this;
    };//end function

    T& operator*(void)              {return *t;}
    const T& operator*(void) const {return *t;}
    T* operator->(void)             {return  t;}
    const T* operator->(void)const {return  t;}
    T* operator()()                 {return  t;}
    T const* operator()() const     {return  t;}

private:
    T * ptr;
    ptr(const ptr<T>&);
    ptr& operator=(const ptr<T>&);
};//end class
```

Beachten Sie dabei, dass im Konstruktor oder im `operator=(..)` die Argumente nicht als `const` deklariert sind. Hier dürfen natürlich keine Zeiger auf Konstante verwendet werden, da diesen je weder neue Inhalte zugewiesen werden dürfen noch später eine Freigabe des Speichers erfolgen darf.

Damit sich ein Trägerobjekt identisch zu einem Zeigerobjekt verhält, sind noch ergänzende Funktionen sinnvoll, beispielsweise ein Vergleichsoperator, um festzustellen, ob auf einem Trägerobjekt eine Zeigervariable abgelegt wurde, sowie Spezialisierungen der `swap`-Funktion:

```
template <typename T>
bool operator==(ptr<T> const& t1, void const* p){
    return t1()==p;
}//end function

template <typename T>
void swap(ptr<T>& t1, ptr<T>& t2){
    swap(t1.t,t2.t);
}//end function
```

```
template <typename T>
void swap(ptr<T>& t1, T*& p){
    swap(t1.t,p);
}//end function
```

Die `swap`-Funktionen sind in der `ptr`-Klasse als `friend` zu deklarieren.

Objekte der Klasse `ptr` können wir innerhalb des Programmbereichs, in dem sie deklariert sind, wie Zeigervariable nutzen. In Funktionsaufrufen bestehen jedoch Einschränkungen: wir können die Objekte als (*konstante*) Referenz übergeben oder mit Hilfe von `operator()` den Zeiger auf das eigentliche Datenobjekt übergeben. In beiden Fällen ist die Verwendung nicht mit der Verwendung normaler Zeigervariabler identisch. Möglich sind natürlich auch weiterhin „Fehlbedienungen“, wie sie mit normalen Zeigervariablen auch auftreten können, wie die Übergabe des Zeigerwertes an eine zweite Trägervariable oder die zusätzlich Speicherung des Zeigerwertes an einer Stelle, an der eine Vernichtung des Objekes nicht bemerkt werden kann. Zumindest für die Belegung zweier Trägerobjekte mit dem gleichen Wert kann man eine Kontrollfunktion implementieren:

Aufgabe. Die Nutzung der Trägerklasse setzt implizit voraus, dass nicht mehrfach die gleiche Zeigervariable verschiedenen Instanzen der Trägerklasse zugewiesen wird:

```
A* a = new A();
ptr<T> p1(a),p2(a);
```

Am Ende des Programms würde nun die Variable `a` zweimal freigegeben, was natürlich nicht funktioniert. Zur Kontrolle könnte man eine Funktion `assert`-Funktion implementieren, die bei Gleichheit der Attributwerte zweier `ptr`-Variablen ein Attribut freigibt und anschließend eine Ausnahme wirft. Implementieren Sie eine solche `assert`-Funktion.

5.3.3 Eine Instanz – mehrere Variable

Eine Referenzzählung ist eine Art objektinterne Buchführung, wie viele Programmteile einen Zugang zu dem Objekt besitzten. Sie wird notwendig, wenn mehrere Zeigervariablen gleichzeitig auf ein Objekt zeigen und der Zeitpunkt des Ungültigwerdens jeder Variablen nicht eindeutig festgelegt ist oder, mit anderen Worten, die Besitzrechte nicht eindeutig festgelegt sind. Sie soll sicherstellen, dass jede dieser Variablen jederzeit auf ein gültiges Objekt zugreifen kann, das Löschen aber auch nicht vergessen wird.

Als Beispiel für Referenzzählungen stelle man sich eine verkettete Liste vor, die Zeigerobjekte enthält, sowie eine Funktion, die Zeiger–Objekte erzeugt und in die Liste einfügt, wobei keine Kopien der Objekte erzeugt werden, sondern die Originale in der Liste gespeichert werden. Anwendungen dieser Art treten bei Pipeline–Verarbeitungen und Fenstertechniken auf dem Bildschirm recht häufig auf. Während die Variable der Erzeugungsfunktion am Ende ihres Deklarationsblockes ungültig wird, kann

- die Liste und das gespeicherte Objekt länger bestehen,
- das Objekt möglicherweise gar nicht in die Liste übernommen werden, wenn die internen Regeln der Liste dies nicht zulassen,
- das Objekt bereits vor Beenden der Erzeugungsfunktion wieder aus der Liste entfernt werden.

Spätestens mit dem Verschwinden beider Verweise muss auch das Objekt gelöscht werden, um Speicherüberläufe zu verhindern; es darf jedoch auf keinen Fall gelöscht werden, so lange noch einer der beiden Verweise benutzt wird. Anstatt nun eine recht komplizierte Ablaufverwaltung des Programms zu implementieren, die die Gültigkeiten beobachtet (*und entsprechend fehleranfällig bei Programmerweiterungen ist*), bietet es sich an, dem Objekt selbst die Aufgabe zu übertragen, darauf zu achten, wann es zerstört werden kann.
Damit es dies machen kann, dürfen im Anwendungsprogramm zwei Vorgänge nicht mehr stattfinden:

(a) Der Zeiger auf das Objekt darf nicht einfach dupliziert werden, sondern das Objekt muss selbst die Zeigerverdopplung vornehmen, um Buch führen zu können.
(b) Der `delete`–Operator darf nicht direkt aufgerufen werden, da das Objekt sonst vorzeitig vernichtet wird. Das Objekt muss eine spezielle Funktion zur Verfügung stellen, in der es selbst über seine Vernichtung entscheidet.

5.3.3.1 Referenzzählung in einer speziellen Trägerklasse

Eine einfache Lösung, die mit allen Zeigerobjekten funktioniert, ist die nochmalige interne Stufung durch ein Zwischenobjekt, das die angerissenen Aufgaben übernimmt. Ohne viele Worte dürfte der folgende Code, den Sie noch entsprechend vervollständigen können, verständlich sein:

```
template <class T> class Ptr_MR {
private:

    template <class U> struct MRHolder {
        U* obj;
        mutable int count;

        MRHolder(){
            obj=new U();
            count=1;
        }//end constructor

        ~MRHolder(){ delete obj; }

        void set(U* p){
            delete obj;
            obj=p;
        }//end function
```

```
        void unref(){
            if(--count==0) delete this;
        }//end function
        MRHolder<U>* ref() const{
            count++;
            return const_cast<MRHolder<U>*>(this);
        }//end function
    };//end class

    MRHolder<T>* mr_obj;

public:
    Ptr_MR() {mr_obj=new MRHolder<T>();}
    Ptr_MR(T* p){
        mr_obj=new MRHolder<T>();
        mr_obj->set(p);
    }//end 'Struktor

    inline Ptr_MR& operator=(const Ptr_MR<T> &p){
        mr_obj->unref();
        mr_obj=p.mr_obj->ref();
        return *this;
    }//end function
    ....
```

Diese Trägerklasse spiegelt die in Java vorhandene Funktionalität wider. Objekte werden – genau betrachtet – immer als Referenzen übergeben, was einige Nebenwirkungen hat. Im Programm

```
typedef Ptr_MR<A> AClass;

void f(A a){ *a= value_2;  }
...
A a;
*a = value_1 ;
f(a);
```

ist der Inhalt auf der enthaltenen Variablen nach Aufruf der Methode `f(..)` `value_2` und nicht etwa `value_1`, wie man bei normalem `call_by_value_` erwarten sollte. Je nach Absicht sollte man daher mit diesen Variablen immer mit normalen oder konstanten Referenzen arbeiten.[7]
Mit dieser Trägerklasse ist auch unser Containerproblem gelöst. Objekte können nun beliebig auf einem Container gespeichert und dennoch anderswo genutzt werden, ohne dass man Gefahr läuft, dass irgendjemand das Objekt löscht und der nächste Nutzer darüber stolpert.

---

[7] Was Java im Übrigen gar nicht kennt, d.h. man erhält bei einer Übergabe eines Objektes an eine Methode von Compiler keine Garantie, dass sich nichts ändert.

Anmerkung. Das Speichermanagement ist auch bei dieser Methode immer noch sehr direkt, d.h. wenn ein Container gelöscht wird, werden alle Objekte entfernt, wenn sonst keine Referenzen darauf bestehen. Die in Java realisierte Garbage-Collection geht etwas anders vor und löscht Objekte erst dann, wenn sie neuen Speicher benötigt, also möglicherweise sehr viel später, als der letzte Objektbezug im Programm verschwindet. Das ist zwar zunächst schneller, ein Nebeneffekt dabei ist aber, dass das Löschen vieler Objekte manchmal dann passiert, wenn man das nicht brauchen kann, weil ein anderer Programmteil dringend Rechenzeit benötigt, sie dann aber nicht bekommt. Überall optimale Universalstrategien existieren auch hier nicht, und man muss für spezielle Anwendungsfälle ggf. selbst Hand anlegen.

5.3.3.2 Referenzzählung durch Vererbung

Obwohl mit der vorstehenden Methode eigentlich alles geklärt, lösen wir die Aufgabe zusätzlich mit Hilfe der Vererbung. Einerseits kann man daran einiges Lernen, welches Potential in Templates wirklich steckt, andererseits eröffnet uns das auch Möglichkeiten, weitere Strategien zu verfolgen.

In einer Basisklasse werden zwei Methoden für die Erzeugung einer neuen Referenz und die Destruktion des Objektes sowie ein Attribut zum Zählen der Referenzen implementiert, wobei das Attribut vom Typ `private` ist, also auch in erbenden Klassen nicht verändert werden kann. Die Methoden umfassen bereits in der Basisklasse die gesamte Funktionalität und sind daher in erbenden Klassen nicht zu überschreiben.

```
class ObjectReferenceCounter {
public:
    ObjectReferenceCounter(){
        multiple_reference_counter=0;
    };//end constructor

    virtual ~ObjectReferenceCounter(){};

    inline ObjectReferenceCounter *
                        NewReference()const{
        multiple_reference_counter++;
        return (ObjectReferenceCounter*) this;
    };//end function

    inline ObjectReferenceCounter * Delete(){
        if(multiple_reference_counter==0){
            delete this;
        }else{
            multiple_reference_counter--;
        }//endif
        return 0;
    };
```

```
    int Instances()const{return
                multiple_reference_counter;};
private:
    mutable int multiple_reference_counter;
};//end class
```

Ihnen wird aufgefallen sein, dass die Löschfunktion den Zeigerwert Null zurückgibt. Der Sinn dieser Aktion wird verständlich, wenn Sie sich an die Regeln für die Zeigernutzung erinnern:

```
// Anwendung:
// ============
ObjectReferenceCounter * a=0, * b=0;
b = new ObjectReferenceCounter();
...
a = b->NewReference();
...
    a=a->Delete();
...
    b=b->Delete();
...
```

Durch die Rückgabe der Null wird die einfache Reinitialisierung der Zeigervariablen ermöglicht, so dass bei Abschluss einer Funktion in der in Kapitel drei beschriebenen Art kontrolliert werden kann, ob alle Objekte hinter Zeigervariablen freigegeben sind.

Das zählende Attribut `multiple_reference_counter` ist mit dem C++-Schlüsselwort `mutable` versehen. Wir haben es schon mehrfach verwendet, erklären es hier aber noch einmal genau: eine neue Referenz auf ein Zeigerobjekt kann ja durchaus in einem Programmbereich erzeugt werden, in dem das Quellobjekt als `const` deklariert ist. Aus diesem Grund ist die Methode `NewReference()` ebenfalls als `const` deklariert, da ja ansonsten ein Compilerfehler die Folge wäre. Die `const`-Deklaration steht jedoch im Widerspruch zur Inkrementierung von `multiple_reference_counter`. Mittels `mutable` wird der Compiler allerdings im Falle dieses einen Attributs „stumm“ geschaltet und erlaubt die Veränderung trotz der `const`-Deklaration. Das Schlüsselwort erlaubt also die Außerkraftsetzung der Compilerprüfungen für bestimmte Ausnahmen und sollte natürlich auch nur dann eingesetzt werden, wenn die Folgen klar sind.

Bliebe noch abschließend zu klären, wie die Basisklasse eingesetzt werden kann. C++ unterstützt das Konzept der Mehrfachvererbung, d.h. die Klasse muss nicht Basisklasse in dem Sinn sein, dass sie die Wurzel der Familie ist. Es ist ach möglich, sie zu einem späteren Zeitpunkt einzbauen:

```
class A { ... };
class B: public A,
```

```
          public ObjectReferenceCounter {...};
class C: public B { ... };
```

Ab der Klasse `B` weist nun auch alle erbenden Klassen die gewünschte Funktionalität auf. Dies macht den Einsatz dieser Technik auch in Klassenbibliotheken möglich, bei denen normalerweise kein Rückgriff auf den Code der Basisklassen besteht und damit auch keine Möglichkeit, die Referenzklasse an der Basis unterzubringen.

Die Mehrfachvererbung kann jedoch auch Nebenwirkungen haben. In der Hierarchie

```
class A: public ObjectReferenceCounter {..};
class B: public A {..};
class C: public A {..};
class D: public A, public B {..}
```

weisen Objekte des Typs `D` zwei Instanzen von `ObjectReferenceCounter` auf, und bei einem Zugriff ist anzugeben, welcher zu verwenden ist:

```
...= d->B::NewReference(..);
```

Derartige Hierarchien können beispielsweise bei der Programmierung graphischer Oberflächen recht schnell entstehen, und die mehrfachen Instanzen der Basisklassen haben dabei meist auch durchaus Sinn, weil die erbenden Teilklassen auf bestimmte individuelle Attribute zurückgreifen müssen. In unserem Umfeld ist dies jedoch fatal, da im Code jeweils eine Spezifikation notwendig ist, was bei einer unterschiedlichen Angabe voraussichtlich zum Systemabsturz führt. Die Vererbung ist daher durch

```
class A: virtual
          public ObjectReferenceCounter {..};
```

zu definieren. Das Schlüsselwort `virtual` an dieser Stelle veranlasst den Compiler, in erbenden Klassen jeweils nur eine Instanz der betreffenden Klasse zu implementieren. Damit hätten wir auch in Umgebungen mit Mehrfachvererbungen eine saubere Ordnung wieder hergestellt.

> **Aufgabe.** Implementieren Sie das obige Beispiel mit Kontrollausdrucken in den Konstruktoren und Destruktoren. Beobachten Sie, welche Struktoren mit oder ohne `virtual` aufgerufen werden.

5.3.4 Mehrfachreferenzen und automatische Verwaltung

Mit `ObjectReferenceCounter` haben wir zwar nun eine Mutterklasse für die Beobachtung des Gültigkeitsbereiches einer Zeigervariablen, allerdings mit folgenden „Nebenwirkungen“:

- Alle Klassen, die diese Funktionalität besitzen sollen, müssen von der Basisklasse erben.

- Der Umgang mit den Zeigern wird völlig umgestaltet und entspricht nicht dem Standardumgang mit Zeigervariablen.
- Wir haben keine Kontrollmechanismen, ob an allen Positionen tatsächlich der gleiche Umgang mit den Zeigervariablen durchgeführt wird.

An der ersten Eigenschaft werden wir nichts ändern, da ein Einbau der Basisklasse an irgendeiner passenden Stelle in einer Klassenhierarchie wohl ohne große Probleme möglich ist.[8] Die beiden anderen Probleme werden wir mit einer Erweiterung der `ptr`-Klasse lösen. Da wir nur eine andere Art benötigen, an bestimmten Stellen mit den Zeigerobjekten umzugehen, können wir dies mit einer `policy`-Klasse als Template-Argument in der Art erledigen, in der auch die STL-Klassen den Umgang mit dem Speicher realisieren. Für die Funktionalität des einfachen Träger sieht dies folgendermaßen aus:

```
template <typename T> class SimplePtr{
public:
    static void Delete(T* t){ if(t) delete t; }
private:
    static T* NewRef(T const*);
};//end class

template <typename T,
          template <typename> class Ref=SimplePtr >
class ptr {
public:
    ptr()      {t=0;}
    ptr(T* p)  {t=(T*)p;}
    ptr(const ptr<T,Ref>& p){
        t=Ref<T>::NewRef(p.t);
    }//end c'tor

    ~ptr(void) {Ref<T>::Delete(t); }

    ptr<T,Ref>& operator=(T* p){
        Ref<T>::Delete(t);
        t=(T*)p;
        return *this;
    }//end function

    ptr<T,Ref>& operator=(const ptr<T,Ref>& p){
        Ref<T>::Delete(t);
        t=Ref<T>::NewRef(p.t);
    }//end function
```

[8] Schließlich bietet C++ Mehrfachvererbung an, womit das Problem tatsächlich an der Stelle einer bestehenden Hierarchie gelöst werden kann, ab der das Mehrfachreferenzkonzept benötigt wird.

```
    T& operator*(void)              {return *t;}
    const T& operator*(void) const {return *t;}
    T* operator->(void)             {return  t;}
    const T* operator->(void)const {return  t;}
    T* operator()()                 {return  t;}
    T const* operator()() const     {return  t;}

private:
    T * t;
};//end class
```

Die `ptr`-Klasse hat zwar nun keine privaten Methoden mehr, man überzeugt sich aber leicht davon, dass die gleiche Funktionalität vorhanden ist, da nun die Methoden in der `policy`-Klasse privat sind und die gleichen Wirkungen haben. Für Mehrfachreferenzen implementieren wir die folgende `policy`-Klasse:

```
template <typename T> class MulRef {
public:
    static void Delete(T* t){t->Delete();}

    static T* NewRef(T const* t){
        return t->NewReference();
    }//end function

private:
};//end class
```

Auch hier weist man nun leicht nach, dass die geforderte Funktionalität nur bei korrektem Erbverhalten realisiert wird und andernfalls mit einem Compilerfehler endet.

> **Aufgabe.** Man kann hier noch zwei Modifikationen vornehmen, die die Trägerklasse etwas weiter von normalen Zeigern entfernt und in der Nutzung etwas sicherer macht. Die erste Modifikation betrifft die Initialisierung, bei der zunächst ein Nullzeiger entsteht. Fehlt eine Initialisierung durch ein mit `new` zugewieses Objekt, kommt es zu Laufzeitfehlern. Alternativ kann man mit einer `Init()`-Methode in der policy-Klasse dafür sorgen, dass immer ein Objekt erzeugt wird und Fehlzugriffe unterbleiben.
>
> Die zweite Modifikation betrifft den direkten Zugriff auf den Zeigerwert mittels `operator()`. Da nun keine Probleme mehr bestehen, `ptr`-Objekte in Funktionsaufrufen zu übergeben, können diese Methoden gesperrt werden.
>
> Implementieren Sie eine Klasse `MulRefResticted`, die dies macht, und ergänzen Sie `SimplePtr` und `MulRef` durch entsprechende Methoden.

Darüber hinaus sind auch `operator==` sowie die `swap`- und `assert`-Funktionen noch ein wenig anzupassen:

```
template <typename T, template <typename> class S>
bool assert(ptr<T,S> const &,ptr<T,S> const &);

template <typename T>
bool assert(ptr<T,Simple> const&,
            ptr <T,Simple> const & ){...}

template <typename T>
bool assert(ptr<T,MulRef> const&,
            ptr<T,MulRef> const & ){...}
```

5.3.5 Zeigerkopien

Anstatt nur mit einem Objekt zu arbeiten und jeweils mehrere Zeiger darauf verweisen zu lassen, kann es ja auch beabsichtigt sein, eine Kopie von Objekten herzustellen. Wir benötigen dann „nur“ eine dritte Policy, die statt einer Referenz eine Kopie erzeugt. Wie das „nur“ bereits andeutet, ist die Sachlage allerdings nicht so einfach.

Das Arbeiten mit Zeigern erfolgt ja vorzugsweise dort, wo Vererbung genutzt wird. Bei einem Kopiervorgang muss es daher nicht ein Objekt des Basistyps sein, von dem eine Kopie hergestellt wird, sondern es kann auch ein Objekt einer beliebigen erbenden Klasse sein. Eine Kopie herzustellen erfordert einen passenden Konstruktor, und das Problem ist, dass die Arbeit mit Konstruktoradressen im Gegensatz zu der mit Methodenadressen aus verschiedenen Gründen nicht zulässig ist.

Wie bei der Referenzzählung bleibt uns daher an dieser Stelle nichts anderes übrig, als einem Objekt die Erstellung einer Kopie selbst zu überlassen. Wir „peppen“ deshalb die `ObjectReferenceCounter`-Klasse etwas auf. Der Aufwand ist allerdings höher als bei der Referenzzählung, da in jeder erbenden Klasse eine Kopierfunktion implementiert sein muss. Dies kontrollieren wir durch ein zweistufiges Modell, das nach Erzeugen eines Objektes kontrolliert, ob es tatsächlich das richtige ist:

```
class PtrBase: public ObjectReferenceCounter {
public:
    ...
    PtrBase* Clone()const;
protected:
    virtual PtrBase* PtrDup()const;
    ...
};//end class

PtrBase* PtrBase:: Clone()const {
    PtrBase* p = this->PtrDup();
    if(typeid(*this)!=typeid(*p)){
```

```
        string s;
        s="Error in class construction of class ";
        s=s+typeid(*this).name();
        s=s+", PtrDup-Function missing";
        delete p;
        throw extended_exception(s);
    }//endif
    return p;
}//end function
```

Die virtuelle Methode `PtrDup()` muss in jeder neu hinzukommenden Klasse neu implementiert werden. Vergißt man dies, erzeugt `Clone()` eine Ausnahme, da die `type_info`-Information des neuen Objektes nicht zu der des Arbeitsobjektes passt. `typeid` () ist ein spezieller Operator, der eine konstante Referenz auf ein Objekt des Typs `type_info` liefert, dessen Klassentyp folgende Schnittstellen bereitstellt:

```
class type_info   {
public:
    virtual ~type_info();

private:
    type_info& operator=(const type_info&);
    type_info(const type_info&);

protected:
    const char *__name;

public:
    const char* name() const;
    bool operator==(const type_info& __arg) const;
    bool before(const type_info& __arg) const;
    virtual bool __is_pointer_p() const;
    virtual bool __is_function_p() const;
    ...
};//end class
```

`type_info`-Methoden können also nur zusammen mit einem `typeid()`-Aufruf verwendet werden und liefern eine Reihe von Informationen über die Typzugehörigkeit des Objektes zur Laufzeit. Um dem Compilerbauer hier ausreichenden Freiraum für Optimierungen zu lassen, ist die Schnittstelle relativ begrenzt.

Aufgabe. Implementieren Sie die `policy`-Klasse `PtrCopy`.

Der einzigeWermutstropfen bei der Angelegenheit ist, dass man erst zur Laufzeit auf einen Implementierengsfehler hingewiesen wird; wie Eingangs beschrieben haben wir diese Möglichkeit aber ausdrücklich vorgesehen und sie kommt nicht unerwartet.

5.3.6 Mischen der Funktionalität, Zulässige Zuweisungen

In einigen Anwendungen kann es wünschenswert sein, zwischen den Speicherstrategien zu wechseln und von einem Mehrfachreferenzobjekt eine Kopie zu erzeugen oder eine Kopie mit mehreren Referenzen zu verwenden. Denkbar sind auch Up- und Downcast-Operationen, d.h. in einer Methode soll ein Objekt mit den Methoden einer erbenden Klasse arbeiten, die in der Basisklasse noch nicht zur Verfügung stehen. Dazu muss das Objekt natürlich einen entsprechenden Typ aufweisen.
Der Programmkode für diese Anforderungen kann beispielsweise folgendermaßen aussehen:

```
class A {...}
class B: public A {...}

void foo(ptr<A> p){
   ptr<B> q;
   try{
      q=p;
      ...
   }catch(...){
      // Inhalt von p ist nicht vom Typ B
   }//endtry
}//end function
```

Ist die Typwandlung möglich, wird im Programm fortgefahren, ansonsten per Ausnahme unterbrochen. Alternativ wäre natürlich auch

```
if((q=p).valid()){ ...
```

mit einer Hilfsmethode `valid()`, die anzeigt, ob die letzte Konversion gültig war. Die Kontrolle muss für das angegeben Beispiel zur Laufzeit unter Verwendung von `dynamic_cast<..=()` erfolgen, um auch folgende Zuweisungen korrekt verarbeiten zu können:

```
class C: public B {..};  // Klasse B wie oben

ptr<A> p1=new C();
ptr<B> p2=p1;
```

Da das auf `p1` gespeicherte Objekt mit der Klasse `B` kompatibel ist, ist eine Umspeicherung möglich (*anders sähe es aus, wenn nur ein Objekt des Typs* `A` *gespeichert ist*). Für die Umspeicherung auf ein anderes `ptr`-Objekt sind folgende Erweiterungen der Klassenschnittstelle notwendig:

```
// Typ der policy-Klasse, Deklaration im
// private-Bereich
typedef Ref<T> Policy;
```

```
// Zuweisungsoperator im public-Bereich
template <typename U, template <typename> class R>
ptr <T,Ref>& operator=(ptr<U,R> const& p);

// friend-Deklaration im private-Bereich
template<typename,template<typename> class >
friend class ptr;
```

Das Innenleben des Zuweisungsoperators hat folgendes Aussehen:

```
template <class U, template <typename> class R>
ptr& operator=(const ptr<U,R>& p){
   cast_valid=false;
   if(static_cast<const void*>(t!
            =static_cast<const void*>(p.t)){
      U* pu=static_cast<U*>
             (ptr<U,R>::Policy::NewRef(p.t));
      T* pp=dynamic_cast<T*>(pu);
      if(pp!=0){
         Ref<T>::Delete(t);
         t=pp;
         cast_valid=true;
      }else{
         ptr<U,R>::Policy::Delete(pu);
         throw ...;
      }//endif
   }//endif
   return *this;
}//end function
```

Die Kontrolle besteht zunächst aus einer `static_cast`-Zeigerprüfung, deren Aufgabe lediglich die Kontrolle ist, ob nicht beide Objekte bereits ohnehin auf das gleiche Objekt zeigen. Anschließend wird eine neue Zeigervariable des neuen Typs erzeugt, die je nach Policy der abgebenden Klasse eine Kopie oder eine weitere Referenz ist. `dynamic_cast` setzt die Zeigerveriable um, liefert aber nur dann einen Zeiger zurück, wenn eine Typwandlung möglich ist, sonst einen Nullzeiger.

Passt die Typwandlung nicht, wird das Zeigerobjekt mittels der erzeugenden Policy wieder vernichtet. Andernfalls wird es auf die andere Trägerklasse übernommen und nun nach einer andere Policy behandelt. Als Konsequenz kann nun ein Zeigerobjekt existieren, auf das mehrere Referenzen existieren, von der einige in Trägerklassen auftauchen, die Kopien statt Referenzen erzeugen. Das ist nun kein Fehler, sondern der Programmierer muss eben genau wissen, was er mit einem Objekt machen will. Die Kopier-Policy darf aufgrund dieser Möglichkeiten aber nun nicht einfach ein Objekt mit `delete` freigeben, sondern muss die `Delete()`-Funktion der `MulRef-policy` verwenden. Kontrollieren Sie, ob Ihre Implementation dies beachtet!

5.3.7 Vollautomatische Policy-Auswahl

Bei der Verwendung der `ptr`-Typen müssen wir jeweils darauf achten, welche Policy mit bestimmten Objekten verbunden ist. Auch wenn das im Einzelfall sicher nicht immer zu vermeiden ist, ist in den meisten Fällen mit einer Klasse eines bestimmten Typs auch eine bestimmte Policy verbunden.

Wir haben in unserem Arbeitsmodell die Steuerklasse für die Mehrfachreferenzen mit der für die Erstellung von Kopien verbunden, es ist aber eine leichte Übung, dies in eine separate Steuerklasse auszulagern (*und auf Klassen, die beide Strategien benötigen, durch Mehrfachvererbung wieder zu übertragen*).

> **Aufgabe.** Trennen Sie die Klasse `ObjectReferenceCounter` wieder auf in die ursprüngliche Klasse und eine Klasse `CloneFactory` für die Kopie-Strategie.

Das würde erlauben, drei Standardtypen festzulegen:

(a) Klassen, die weder von `ObjectReferenceCounter` noch von `CloneFactory` erben, sind mit der Policy `Simple` zu verarbeiten.
(b) Klassen, die von `ObjectReferenceCounter` erben, sind mit `MulRef` zu verarbeiten.
(c) Klassen, die von `CloneFactory` erben, sind mit `PtrCopy` zu verarbeiten.

Um dies durch eine Automatik zu realisieren, ist eine Methode zur Erkennung von Vererbungsbeziehungen zur Compilezeit notwendig. Im Prinzip ist das recht einfach, denn eine Zuweisung von Zeigern der Art

```
A* a; B* b;
a=b;
```

ist nur zulässig, wenn `B` von `A` erbt. Allerdings darf diese Prüfung nicht dazu führen, dass der Compiler seine Arbeit beendet. Mit Hilfe von Templates und Polymorphie ist ein Lösung jedoch recht einfach möglich.

```
template <typename A, typename B>
class FirstInheritsFromSecond {
private:
    typedef char One;
    typedef struct {char a[2];} Two;
    static One Test(B*);
    static Two Test(...);
    static A* Make();
public:
    enum { yes=(
sizeof(FirstInheritsFromSecond<A,B>::Test(Make()))
                    ==sizeof(One)) };
};//end class
```

Die Klasse soll feststellen, ob der erste Templateparameter vom zweiten erbt, und als Ergebnis auf dem enum-Wert `yes true` oder `false` aufweisen. Die zentrale Instanz dazu stellt die Methode `Test(..)` dar, die in zwei Formen deklariert ist. Gemäß dem C++-Sprachstandard muss der Compiler bei einem Aufruf polymorpher Funktionen diejenige Variante wählen, die am besten zu dem Aufruf passt. Die erste Variante erwartet einen Zeiger des Typs `B`, und sofern `A` von `B` erbt, ist dieser Aufruf zulässig und auch die beste Lösung. Erben die Klassen nicht voneinander oder ist die Vererbungsreihenfolge anders herum, ist der Aufruf unzulässig und der Compiler wählt die zweite Variante, die jeden Parameter akzeptiert. Beide Funktionen besitzen unterschiedliche Rückgabetypen, an denen erkannt werden kann, welche Funktion vom Compiler ausgewählt wurde.

Das Problem, nun tatsächlich einen Zeiger des Typs `A` für die Prüfung präsentieren zu müssen, wird durch die Funktion `Make()` erledigt, die per Deklaration einen Rückgabewert dieses Typs aufweist. Der Rückgabetyp von `Test(..)` wird wiederum mittels der sizeof-Funktion überprüft, die die Größe eines Typs oder eines Objekts liefert. Aufgrund der ersten Eigenschaft kann sie dies auch bereits während des Compilierens. Erbt also nun `A` von `B`, so wird im Funktionsaufruf des enum-Typs die erste Variante von `Test(..)` implementiert, die einen Rückgabewert des Typs `One` besitzt und damit die logische Auswertung auf `true` erzwingt.

Das Bemerkenswerte an der Prüfung ist, das alles während der Compilierung abgewickelt wird, denn der `enum`-Typ muss ja für die Erstellung einer lauffähigen Anwendung komplett definiert sein. Die verwendeten Funktionen besitzen nur Schnittstellendeklarationen, aber keinerlei Implementierung, d.h. das Ganze verursacht keinen Speicher- oder Laufzeitaufwand in der Anwendung.

Wie ist nun damit die Automatisierung der Policy-Wahl zu erreichen? Wir implementieren dazu die Klasse `AutoPolicy`, die die gleiche Schnittstelle wie `Simple` ff. besitzt, intern aber auf eine der bereits bestehenden Policy-Klassen zurückgreift. Dazu erweitern wir zunächst unsere Vererbungsprüfung:

```
template <typename T> struct InheritFromPolicy{
 enum { pnr =
   FirstInheritsFromSecond <T,MulRef>::yes +
   2*FirstInheritsFromSecond <T,CloneFactory>::yes};
};//end class
```

Als Ergebnis hat `pnr` den Wert 0, wenn von keiner der Policyklassen geerbt wird, 1 bei Mehrfachreferenzen, 2 bei Kopierstrategie und 3 bei beiden (*bei der letzten Möglichkeit muss man sich dann überlegen, welche Strategie Vorrang besitzt*). Über Template-Spezialisierungen gelingt dann die Auswahl der Policy-Klasse:

```
template <typename T> class AutoPolicy {
private:
  template <class U,int> struct Pselect;

  template <class U> struct Pselect<U,0> {
    typedef Simple<U> SPolicy;
  };
```

```
  template <class U> struct Pselect<U,1> {
    typedef MulRef<U> SPolicy;
  };

  template <class U> struct Pselect<U,2> {
    typedef CloneFactory<U> SPolicy;
  };

  typedef typename
     Pselect<T,InheritFromPolicy<T>::pnr>::SPolicy
          policy;
public:
    static void Delete(T* t){
       policy::Delete(t);}

    static T* NewRef(T const* t){
        return policy::NewRef(t);
    }//end function

};//end class
```

Die innere Klasse `Pselect` existiert in 3 Spezialisierungen, die jeweils unterschiedliche Policies als Typen abliefern. Mit Hilfe der Vererbungsprüfung wird nun eine der `Pselect`-Klassen ausgewählt und der Policy-Typ damit verbindlich festgeschrieben. Das Syntaxelement `typename` nach `typedef` ist notwendig, weil der Compiler zunächst die Syntax formal prüft, ohne eine explizite Instanziierung mit einer Klasse `T` aber aus dem Term `Pselect<T,InheritFromPolicy<T>::pnr>::SPolicy` zunächst nicht entnehmen kann, ob es sich tatsächlich um einen Datentyp handelt (*es könnte ja auch* `enum` *oder ein Attribut dahinter stecken. Um das festzustellen, muss* `Pselect` *schon mit einem bestimmten Templateparameter instanziiert werden, wobei dann auch alle Spezialisierungen aufgelöst werden. Nach der Instanziierung durch ein Objekt ist das natürlich klar*). Der C++-Standard schreibt daher vor, dass an diesen Stellen der Programmierer durch die zusätzliche Angabe von `typename` die Angelegenheit klarstellen muss. Auch hier muss man sich wieder vor Augen halten, dass der nicht ganz triviale Schreibaufwand ausschließlich für die automatische Auswahl während der Übersetzung benötigt wird. Das Ergebnis ist zur Laufzeit zum Nulltarif zu haben.

5.4 Steuerung der Ausnahmebehandlung

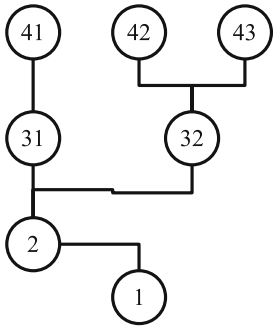
5.4.1 Anforderungen an die Ausnahmesteuerung

Wir wollen nun genauer untersuchen, wie Ausnahmen zur Behandlung seltener aber gleichwohl in einem Prozessablauf gut beschriebener Situationen eingesetzt werden können. Bei den seltenen Situationen handelt es sich meist um Störungen

oder „Fehler“ im normalen Ablauf. Die Reduktion auf den Begriff „Fehler“ ist dann schnell bei der Hand, und die hiervon ausgehende Konstruktion von Beispielen führt dann zu dem etwas verzerrten Bild von Ausnahmen.[9] Als Beispiel für eine seltene aber vorgesehene Situation denken Sie an eine komplexe mehrstufige Anwendungsmaske, die vom Anwender und vom Programm interaktiv ausgefüllt werden soll. Entschließt sich einer der beiden Akteure irgendwann dazu, den Vorgang abzubrechen, weil planmäßig erst im Verlauf des gesamten Verfahrens festgestellt werden kann, ob es zu Ende zu bringen ist, so ist das ein völlig rationaler Vorgang und hat mit „Fehlern“ sicher nichts zu tun (*falls Ihnen das zu abstrakt ist: fast jeder hat vermutlich schon mal eine Maske im Internet ausgefüllt, bei der erst recht spät auffällt, dass ein Fortfahren etwas kostet. Wenn man dazu aber nicht bereit ist, muss ein Rückstellen als zulässige Option definiert sein*). Die „planmäßige“ Berücksichtigung von Abbruchbedingungen zusammen mit dem Rückfahren des Programms in einen „Normalzustand“ (*etwa das Löschen bereits erfasster Daten*) kann aber so aufwendig sein, dass die Übersicht verloren geht und die Anwendung fehleranfällig wird. Das Werfen einer Ausnahme und die Durchführung des Rückspulvorgangs in der Ausnahmebehandlung kann da wesentlich ökonomischer sein. Wir werden das weiter unten an einem Anwendungsbeispiel im Detail erläutern und hier zunächst ein Universalwerkzeug zur Abwicklung komplexer Vorgänge während der Ausnahmebehandlung entwerfen.

Stellen wir zunächst unsere Anforderungen an ein Ausnahmebehandlungsmanagement zusammen. Wir betrachten Anwendungen mit einiger Verzweigungsbreite und Verschachtelungstiefe an Funktionen. Das Programm durchlaufe nacheinander die Funktionen

```
1->C(2)->C(31)->C(41)->R(31)->R(2)->C(32)->C(42)
   ->R(32)->C(43)->R(32)->R(2)->R(1)
```



Aufrufmodell für ein Ausnahmemanagement

[9] Man kann sich das vielleicht auch so vorstellen: Die Übersetzung von Hamlets Worten „to be or not to be...“ ins Deutsche ist ja bekanntlich „sein oder nicht sein..“. Reißt man das aus dem Zusammenhang und lässt es Rückübersetzen, sind durchaus Ergebnisse der Art „his or not his..“ denkbar. Auf ähnliche Weise ist möglicherweise die Reduktion auf den Begriff „Fehlerbehandlung“ abgelaufen.

Dabei bedeute C(..) den Aufruf einer Unterfunktion, R(..) die Rückkehr in die Oberfunktion. In jeder Funktion können prinzipiell Ausnahmesituationen eintreten, in denen bestimmte Aktionen durchgeführt werden müssen. Eine Liste der Aktionen wird zusammen mit den notwendigen Informationen in einem speziellen „Ausnahmebehandlungsobjekt“ hinterlegt, das bei Eintreten der Ausnahmesituation geworfen und bei Ausbleiben der Situation normalerweise ohne Durchführung irgendeiner Aktion wieder gelöscht wird.

Abweichend davon sind aber durchaus Anwendungen denkbar, in denen die notierten Aktionen bei Ausnahmen auch bei Ausbleiben einer Ausnahmebedingung während der Bearbeitung des aktuellen `try`–Blockes bei Eintritt einer Ausnahmebedingung zu einem späteren Zeitpunkt in einem ganz anderen Arbeitszusammenhang abgearbeitet werden müssen. In diesem Fall dürfen die Ausnahmebehandlungsobjekte bei Verlassen ihres Deklarationsblockes nicht gelöscht werden!

Betrachten wir dazu das Beispiel in der nebenstehenden Grafik. Die in (31) und (41) definierten Aktionen sollen ihre Gültigkeit behalten, wenn bis einschließlich des Ablaufs der Funktion (42) Ausnahmen auftreten. In (42) werde nun eine Ausnahme geworfen, die zunächst in der Funktion (32) gefangen wird. Dabei werden alle Objekte in (42) und alle Objekte im `try`–Block von (32) ungültig, und die im geworfenen Objekt hinterlegten Ausnahmeaktionen werden im Standardverfahren ausgeführt. Anschließend werden alle weiteren in zuvor entsprechend markierten Objekten aus (32), (42), (31) und (41) hinterlegte Aktionen durchgeführt. Die Ausnahmebehandlungsobjekte haben nun ihren Dienst verrichtet und werden gelöscht.

Betrachten wir nun den Fall, dass (42) nach (32) zurückspringt und nun (43) aufgerufen wird. Die Ausnahmeaktionen von (42) sowie von (31) und (41) sollen dabei ungültig werden, das heißt wirft nun (43) eine Ausnahme, die in (32) gefangen wird, so ergibt sich ein völlig anderes Ausführungsbild. Die gesicherten Ausnahmebehandlungsobjekte müssen daher vor dem Aufruf der Funktion (43) entfernt werden.

Grundsätzlich ändert sich an diesen Abläufen nichts, wenn die Funktion (32) für die Ausnahmebehandlung gar nicht zuständig ist, sondern die Ausnahme an die Funktionen (2) oder (1) weiter wirft. Wenn keine Ausnahmesituation eintritt (*und das sollte ja in der überwiegenden Anzahl der Anwendungsfälle der Fall sein*), müssen die gesicherten Objekte entfernt werden. Abgesehen von dem Unfug, der bei einer späteren Ausführung im Rahmen einer Ausnahmesituation, die nichts mit der Sicherungssituation zu tun hat, geschehen kann, kommt ein Versäumnis der Produktion eines Speicherlecks gleich, da immer mehr unnötige Objekte gesammelt werden.

Das Modell wird durch die Bedingung, Ausnahmebehandlungsvorschriften auch ohne Ausnahmesituation über ihren Deklarationsblock hinaus ausführungsfähig zu halten, schon recht komplex und ähnelt den strategischen Planungsspielen von Militärapparaten: Während die Regierung (*das Programm innerhalb der* `try`*–Blöcke*) „service as usual“ betreibt, werden im Hintergrund Planungen für den Notfall (*Ausnahmebehandlungsobjekt*) durchgeführt und archiviert (*das ist noch zu realisieren*).

Im Notfall (*Werfen der Ausnahme*) werden die Pläne ausgeführt, läuft alles ohne Probleme ab, können die Notfallpläne irgendwann vernichtet werden (*auch das ist noch zu realisieren*).
Wir ergänzen es noch durch eine letzte Bedingung: Ausnahmebehandlungsobjekte können aktiviert und deaktiviert werden. Sind sie deaktiviert, so wird selbst bei Vorliegen einer Ausnahmesituation keine Ausnahme geworfen. Um das an der Grafik zu verdeutlichen, stellen Sie sich vor, dass die Funktionen (41) und (42) identisch sind, allerdings nur beim Aufruf (32) -> (41) Ausnahmen geworfen werden dürfen. Da die Funktion (41) nicht weiß, in welchem Umfeld sie arbeitet (*wenn von einer Erweiterung der Parameterliste der Funktion einmal abgesehen wird*), muss die Aktivierung oder Desaktivierung in der rufenden Funktion vorgenommen werden.

5.4.2 Implementation 1: Realisierung der Objektleitung

Um solche komplexen Arbeitsregeln befolgen zu können, setzen wir die im letzten Kapitel eingeführte automatische Zeigerverwaltung mit speziellen Ausnahmeklassen ein. Für die Durchführung der Ausnahmebehandlung definieren wir zunächst die Klasse `exception_b`, die von `ObjectReferenceCounter` erbt. Die Ablaufverwaltung erfolgt mit Zeigerobjekten dieser Klasse, die einen Informationstransport in die `catch`–Blöcke erlauben. In welcher Form die Informationen transportiert werden, untersuchen wir weiter unten. Wir statten die Klasse mit einer Reihe von Attributen aus, die für die Aktivierung/Desaktivierung und Sicherung benötigt werden:

```
class exception_b: public ObjectReferenceCounter {
protected:
    int  _group,         // externe Steuerung der
        _id;             // Aktivität eines Objektes
    bool _enabled,       // lokale aktiv/inaktiv
         _permanent;     // Sicherungsvermerk
public:
    exception_b();
    virtual ~exception_b();
    void Throw();
    void Catch(int group, int ident);
};//end class
```

Die Attribute `_group` und `_id` erlauben die Zusammenfassung von Ausnahmebehandlungsobjekten in Gruppen und innerhalb der Gruppen eine individuelle Kennung jeder einzelnen Ausnahmebedingung. Sie werden bei der globalen Steuerung zur Aktivierung/Desaktivierung von Ausnahmen oder zum Entfernen von gesicherten Objekten eingesetzt. Das Attribut `_enabled` erlaubt das lokale Aktivieren/Deaktivieren eines Objektes und wird benötigt, falls Steuerungsvorgänge durch Übergabeparameter realisiert werden oder das Ausnahmebehandlungsobjekt

selbst als Übergabeparameter in einem Funktionsaufruf auftritt. Das Attribut `_permanent` gibt an, ob das Objekt bei Verlassen des Deklarationsbereiches gesichert werden soll oder nicht.

> **Aufgabe.** Implementieren Sie Bedienmethoden für die Attribute sowie den Konstruktor. Initialisieren Sie die Objekte so, dass das normale Ausnahmebehandlungsschema abläuft.

Ob im Falle einer Ausnahme ein Objekt geworfen werden soll oder ein geworfenes Objekt bei einem Fang an der richtigen Stelle angelangt ist, ist keine simple Entscheidung. Wir überlassen diese Entscheidungen den Objekten selbst und deklarieren dazu die beiden Methoden `Throw()` und `Catch()` . Bevor wir uns deren Innenleben zuwenden, erledigen wir zunächst die automatische Verwaltung von Zeigerobjekten der Ausnahmeklasse. Wir definieren dazu die Klasse `Ausnahme`, die eine Spezialisierung von `Ptr<exception_b>` ist. Sie erhält die Aufgabe, bei Verlassen des Deklarationsbereiches zu entscheiden, ob ein Zeigerobjekt vernichtet werden darf oder gesichert werden muss.

```
class Ausnahme: public Ptr <exception_b> {
private:
    friend exception_b;
    Ausnahme();
    Ausnahme(const exception_b* p);
public:
    Ausnahme(long group, long ident);
    Ausnahme(const Ausnahme& a);
    virtual ~Ausnahme();
};//end class
```

> **Aufgabe.** Implementieren Sie zunächst die Konstruktoren der Klasse.

Erzeugt werden im Normalfall in den Anwendungen nur Objekte des Typs `Ausnahme`, die für die spezielle Ausnahmebehandlung konfiguriert werden. Das hat die Konsequenz, dass zu jedem `try`–Block nur ein `catch`–Block implementiert werden kann, da ein Objekt eines bestimmten Typs nur einmal gefangen werden kann. Wir nutzen dies zur bequemen Implementierung von Ausnahmebehandlungen:

```
#define EXCEPTION_BEGIN(group,id,message)\
    try{\
        Ausnahme ausnahme(group,id);\
        ausnahme->SetMessage(message);

#define EXCEPTION_CATCH(group,id)\
    }catch(Ausnahme ausnahme){\
            ausnahme->Catch(group,id);\
```

```
#define EXCEPTION_END  }

// Implementation
// ==============

void function(..){
    ...
    EXCEPTION_BEGIN(14,23,"Meldungstext")
        Ausnahme a2(14,22), a3(14,27); // optional
        ...    // hier steht der Anwendungskode
        if(...) ausnahme->Throw();
        ...
    EXCEPTION_CATCH(14,-1)
        ...    // hier kommt meist nichts mehr hin
    EXCEPTION_END
    ...
}//end function
```

Sofern Sie auf das Werfen von Objekten anderer Klassen verzichten, muss an diesem Implementationsschema nichts mehr geändert werden. Jedes Ausnahmeobjekt wird am Erzeugungsort mit einer bestimmten Gruppen- und Individualidentität versehen. Gemäß unserer Voraussetzung, nur geplante Aktionen zu bearbeiten, entsprechen die Erzeugungsorte bestimmten Punkten im Ablaufprotokoll des Gesamtvorgangs, so dass die Identitäten eindeutig durch externe Vorgaben definiert und widerspruchsfrei sind.[10] Zusätzlich zu dem durch das Makro deklarierten Ausnahmebehandlungsobjekt können beliebig viele weitere deklariert werden. Zu den bereits vorhandenen Attributen ist hier noch ein Stringattribut hinzu gekommen, das erste Informationen über die Ausnahme enthält. Je nach Notwendigkeit werden die einzelnen Objekte aktiviert oder deaktiviert beziehungsweise permanent oder nicht permament geschaltet (*nicht dargestellt*). Tritt im Programmablauf eine Situation auf, in der das Werfen einer Ausnahme sinnvoll sein könnte, so wird die `Throw()`–Methode des entsprechenden Ausnahmebehandlungsobjektes aufgerufen. Diese entscheidet, ob nun tatsächlich eine Ausnahme geworfen oder das Programm fortgesetzt wird (*eigentlich trivial: Der Programmkode muss natürlich robust gegen eine Fortsetzung nach einem Ausnahmegrund sein*). Eine geworfene Ausnahme wird durch den nächsten `catch`–Block gefangen. Im Makro–Aufruf ist festgelegt, für welche Gruppen und Individuen er zuständig ist, und das Makro gibt dem gefangenen Objekt dies mittels der `Catch(..)`–Methode bekannt. Das Objekt darf ausgeführt werden, wenn Gruppen- und Identifikationsnummer mit den Parametern übereinstimmen. Weist einer der Parameter einen negativen Wert auf, so darf die Ausführung unabhängig vom eigenen Wert erfolgen. Andernfalls wird das Objekt weiter geworfen.

---

[10] Das kann allenfalls durch die gemischte Verwendung von Bibliotheken unterschiedlicher Produzenten unterlaufen werden. In unserem Gesamtmodell ist so etwas aber äußerst unwahrscheinlich.

Mit oder ohne Wurf einer Ausnahme wird irgendwann der Deklationsbereich der Ausnahmebehandlungsobjekte verlassen und dabei über eine Sicherung entschieden. Bei der Behandlung einer Ausnahme werden diese Objekte ausgeführt. Für die Sicherung und Aktivierungskontrolle werden im Exception–Modul zwei Variable und in der Klasse `Ausnahme` einige Bedienmethoden deklariert. Die Implementation dieser und einer Reihe anderer Methoden kann ich nun direkt als Aufgaben formulieren.

```
static deque <pair<int,int> > disabled;
static deque<Ausnahme> work_list;

class Ausnahme ... {
    ...
    static void DisableObject(int group, int obj);
    static bool EnableObject(int group, int obj);
    static void EraseObject(int group, int obj);
    static bool ObjectActive(int group, int obj);
    static void ClearAll();
};//end class
```

> **Aufgabe.** Die Methoden `DisableObject(..)` und `EnableObject(..)` fügen Einträge in die Variable `disabled` ein beziehungsweise löschen diese. Ein Objekt beziehungsweise eine Gruppe von Objekten gilt als deaktiviert, wenn sie in der Liste eingetragen sind. Mit der Methode `ObjectActive(..)` wird festgestellt, ob ein Objekt oder eine (*komplette*) Gruppe aktiv ist. Implementieren Sie die Methoden.
>
> **Aufgabe.** Die Methode `EraseObject(..)` löscht Objekte aus dem Container `work_list`. Bei Gruppen- oder Identifikationsnummern mit negativen Werten sollen alle Objekte mit übereinstimmender positiver Nummer gelöscht werden. Sind beide Kennziffern negativ, so werden alle notierten Objekte gelöscht (*generelles Aufräumen*). Die Sperrlisten bleiben unberührt. Implementieren Sie die Methode.
>
> **Aufgabe.** Die `Throw(..)`–Methode darf nur aktive Objekte werfen. Der Destruktor der Klasse `Ausnahme` muss Objekte mit einer permanent–Kennzeichnung im Container sichern. Implementieren Sie nun auch diese Methoden.

Wir müssen uns nun noch die Methode `Catch(..)` vornehmen. Diese prüft die Gruppen- und Identifikationsnummer und führt bei positivem Prüfergebnis die notwendigen Aktionen durch, die wie einstweilen hinter der Methode `ExecuteFunction()` verstecken. Anschließend werden alle gesicherten Objekte ausgeführt und aus dem Container gelöscht. Bei negativer Prüfung wird die Ausnahme weiter geworfen.

```
void exception_b::Catch(int group, int ident){
    if((ident==_id || ident==-1) &&
       (group==_group || group==-1)){
```

```
        ExecuteFunction();
        while(!work_list.empty()){
            work_list.top()->ExecuteFunction();
            work_list.pop();
        }//endwhile
    }else{
        throw;
    }//endif
}//end function
```

Aufgabe. Entwerfen Sie nun ein Implementationsmodell für Fehler: Referenz nicht gefunden gemäß der Beschreibung im Text. Führen sie Tests durch. Sowohl bei Werfen von Ausnahmen als auch bei normaler Bearbeitung muss die Arbeits- und die Sperrliste am Ende des Programms vollständig leer sein.

Es bleibt noch zu untersuchen, was sich hinter der Methode `Execute Function()` verbirgt. Bei der Definition der Implementationsmakros für das Ausnahmemanagement wurde ein Text deklariert, der an das Ausnahmebehandlungsobjekt übergeben wird. Man könnte zunächst annehmen, dass hier alle Informationen für die Ausnahmebehandlung hinterlegt werden, doch das ist sicher eine zu grobe Lösung. Hier können lediglich ergänzende Informationen hinterlegt werden. Was wirklich bei einer Ausnahme zu geschehen hat, wird durch die an der normalen Ausführung beteiligten Objekte vorgegeben. Sind Aktionen zurückzunehmen, so sind dort in der Regel entsprechende Methoden vorgesehen, die nun aktiviert werden müssen. Dafür sind zwei Voraussetzungen notwendig: Die Objekte müssen zum Zeitpunkt der Ausnahmeausführung noch existieren und es müssen Funktionen bekannt sein, die aufgerufen werden sollen. Bezüglich der Objekte fordern wir nur, dass sie Zeigerobjekte sind und von `ObjectReferenceCounter` erben, damit ihr Lebenszyklus durch die Ausnahmebehandlungsobjekte kontrolliert werden kann. Sie werden im Ausnahmefall an eine spezielle Funktion übergeben, in der die Anweisungen für die Behandlung der Ausnahme hinterlegt sind. Von der Funktion wird die Adresse im Ausnahmebehandlungsobjekt gesichert. Die notwendigen Schnittstellenerweiterungen sind damit:

```
class exception_b: public ObjectReferenceCounter {
public:
    typedef
       vector<APtr<ObjectReferenceCounter> > ACont;
    typedef void (*func)(ACont& a, string s);

    void SetCatchFunction(func f);
    void PushObject(ObjectReferenceCounter* ob);
    void ClearObjects();

    void ExecuteFunction();

protected:
    friend class Ausnahme;
```

```
    func function;
    ACont acont;
};//end class
```

Die Objekte werden in einem Container gesammelt, wodurch deren Anzahl nicht begrenzt wird. Die Ausnahmebehandlungsfunktion erhält diesen Container und den Informationsstring als Übergabeparameter. Container und Funktionsadresse können während der Bearbeitung der Anwendung beliebig verändert werden. Ob die richtigen Objekte in der korrekten Reihenfolge übergeben wurden, kann mit Hilfe von dynamischen `Cast`–Anweisungen in der Funktion überprüft werden.

Aufgabe. Implementieren Sie die restlichen Funktionen für die Ausnahmebehandlung. Für einen Test können Sie folgendes Beispiel implementieren: Ein zentrales Lager verwaltet zwei Materialdepots, aus denen Teile entnommen oder in denen Teile deponiert werden können. Jedes Teil hat einen bestimmten Preis. Zugelassen sind eine Reihe von Kunden, für die Konten geführt werden. Pro Transaktion kann ein Kunde ein Depot, eine Stückzahl und Entnahme oder Deponierung angeben. Folgende Fälle sind möglich:

(a) die Transaktion ist vollständig abwickelbar,
(b) die angeforderte Stückzahl ist nicht vorhanden,
(c) das Konto des Kunden weist keine Deckung bei Entnahme auf,
(d) das Konto des Lagers weist keine Deckung für eine Deponierung auf.
In den Fällen (b) – (d) soll nichts unternommen und der Transaktionsauftrag in eine Liste zur späteren Bearbeitung gestellt werden. Führen Sie (a) durch und verwenden Sie das Ausnahmemanagement, um in den Fällen (b) – (c) (*die bereits abgewickelten Teile von*) (a) rückgängig zu machen.

5.4.3 Implementation II: Mischen von Strategien

Das zweite Modell, das wir hier entwerfen wollen, entspricht mehr dem klassischen Ausnahmemodell, und soll Probleme berücksichtigen, die im Rahmen der Entwicklung von Bibliotheken auftreten. Bei einer Ausnahmesituation kann der Anwendungsprogrammierer die Behandlung im Rahmen des normalen Ablaufs mit Hilfe eines logischen Rückgabewertes oder mit Hilfe einer Ausnahme vorsehen. Der Bibliotheksentwickler muss sich auf beides einrichten und kann eine Ausnahme erzeugen und anschließend einen logischen Rückgabewert ausgeben; die Entscheidung mittels Compilerschalter „Ausnahme an-aus“, was nun tatsächlich ausgeführt wird, ist allerdings grob und erlaubt keine Differenzierung. Sinnvoll wäre eine „Buchführung“, für welche Objekte Ausnahmebehandlungsmethoden implementiert sind, und eine Kontrolle vor dem `throw`-Befehl.

Wir beginnen mit den Typen, die in einer Ausnahmesituation geworfen werden können. C++ stellt für die Behandlung von Ausnahmen die Klasse `exception` zur Verfügung, die als Basisklasse für diejenigen Objekte gedacht ist, die in Ausnahmen geworfen werden. Die Funktionalität ist recht bescheiden; die Klasse stellt mehr

oder weniger nur eine virtuelle Methode `what()` zur Verfügung, die einen String mit einer Nachricht ausgibt. Die Nachricht selbst muss man aber bereits in einer erbenden Klasse erzeugen. Wir sehen nun drei Arten von Objekten, die in einer Ausnahme geworfen werden können, vor:

(a) Werfen eines `exception`-Objekts, das eine Nachricht ausgeben kann.
(b) Werfen eines `exception`-Objekts, das ein anderes beliebiges Objekt transportiert.
(c) Werfen eines Objektes eines beliebigen Typs.

Für den Fall (a) entwerfen wir eine einfache, von `exception` erbende Klasse, die in Fall (b) weiterverwendet wird:

```
struct NullType {};
class extended_exception: public exception {
public:
    extended_exception(string s) throw();
    extended_exception(const extended_exception& e)
                                        throw();
    ~extended_exception() throw();
    extended_exception& operator=(
              const extended_exception& e) throw();
    const char *what ()const throw();
    virtual const type_info& object_type() const
                                            throw();
    virtual void ThrowObject() const;
private:
    extended_exception() throw();
    string reason;
};//end class
```

Für die korrekte Bearbeitung der unterschiedlichen Objekttypen sehen wir nur zwei Methoden vor: `object_type()` liefert Informationen zum transportierten Objekt, `ThrowObject()` erlaubt das Weiterwerfen des transportierten Objektes. Auf dieser Ebene der Klassenhierarchie ist natürlich noch kein transportiertes Objekt vorhanden, was wir mit der Klasse `NullType` überdecken müssen:

```
const type_info& extended_exception::object_type()
                                        const throw(){
    return typeid(NullType);
}//end function

void extended_exception::ThrowObject() const {
    Throw<NullType>(NullType());
}//end function
```

Ein anwendungsspezifisches transportiertes Objekt tritt erst in der Erweiterung (b) auf:

```
template <typename T = NullType>
class managed_exception: public extended_exception {
    typedef select_type<T,true> d_type;
pubic:
    managed_exception(string s, const T& obj=T())
                              throw()
        : extended_exception(s) { t=obj; }
    ...
    const type_info& object_type() const
                              { return typeid(T);}

    void ThrowObject() const {
        if(exception_allowed(typeid(d_type)))
            throw t;
    }//end function

    T& GetObject() { return t;}
private:
    managed_exception() throw();
    T t;
};//end class
```

Die Klasse `managed_exception` ist eine Template-Klasse, d.h. sie wird vom Compiler für jeden Typ eines zu transportierenden Objekts individuell typisiert. Für das Fangen von Ausnahmen sind deshalb auch entsprechende `catch`-Blöcke einzurichten, wenn man eine Ausnahme nicht über den Basistyp `extended_exception` fangen will.

In der Methode `ThrowObject()` für das Weiterwerfen des transportierten Objektes wird die Strategie sichtbar: das Werfen wird nur dann durchgeführt, wenn für den Objekttyp eine Erlaubnis besteht. Diese ist mit einem speziellen Aufbau des `try-catch`-Blocks verbunden:

```
try{
   NotifyHandler<T> n1;
   ...
}catch(managed_exception<T>& exc){
   ...
}//endtry
```

Beim Aufbau eines `try-catch`-Blockes ist klar, welche Objekttypen in den `catch`-Anweisungen gefangen werden können. Um die entsprechenden Ausnahmen freizugeben, wird im `try`-Block ein Objekt von `NotifyHandler<T>` erzeugt, dessen Aufgabe die Registrierung des zu fangenden Objekttyps ist. Nur bei registrierten Typen liefert die Methode `exception_allowed(..)` den Wert `true` zurück und ermöglicht das Werfen einer Ausnahme. Wird der `try`-Block fehlerfrei abgearbeitet, so löscht der Destruktor des Notifier-Objektes die Registrierung wieder und sperrt weitere Ausnahmen dieses Typs. Die Methode ist zwar keine

Vollautomatik – es müssen passende Notifier-Objekte und `catch`-Blöcke vom Programmierer angelegt werden, ermöglicht jedoch die Freigabe von Ausnahmen nach Bedarf.

Die Konstruktion der Notifier-Klasse muss die Objekttypklassen a)-c) unterstützen und außerdem wahlweise eine Ausnahme zulassen oder unterdrücken (*d.h. das Werfen bereits registrierter Ausnahmen kann zeitweise wieder unterdrückt werden*). Benötigt werden daher drei Template-Parameter:

```
template <class T=extended_exception,
          bool direct=false,
          bool allow=true>
class NotifyHandler {
    typedef select_type<T,direct> hn_type;
    enum { value=allow };

public:
    NotifyHandler() throw()  {
        handle_exceptions<allow>::
                            reg(typeid(hn_type));
    }//end function
    ~NotifyHandler() throw() {
        handle_exceptions<allow>::
                           unreg(typeid(hn_type));
    }//end function
};//end class
```

Neben der Typklasse des zu werfenden Objektes werden zwei boolsche Parameter benötigt, von denen der erste spezifiziert, ob ein Objekt der Typklasse direkt geworfen wird oder als spezielle Instanz von `managed_exception`, der zweite die Freigabe oder Sperrung bewirkt. Die Registrierung erfolgt mittels zweier Hilfsklassen. `select_type` bildet aus der Typklasse und einem der boolschen Parameter die zu werfende Klasse:

```
template <class T, bool> struct select_type {
    typedef managed_exception<T> exc_type;
};//end struct

template <class T> struct select_type<T,true> {
    typedef T exc_type;
};//end struct

template <> struct select_type<extended_exception,false> {
    typedef extended_exception exc_type;
};//end struct
```

Diese Hilfsklasse wird auch bereits in `managed_exception` bei der Überprüfung der Freigabe verwendet, obwohl dazu vordergründig eigentlich keine Notwendigkeit besteht, denn der zu prüfende Typ liegt bereits fest. Der Grund hierfür liegt in

der Verwendung des `typeid()`-Operators für die Typkontrolle. Wie bereits oben erwähnt räumt der Operator dem Compiler-Bauer relativ große Freiheiten in der Benennung von Klassen ein, was dazu führen kann, dass in `managed_exception` beim direkten Einsetzen des Typs ein anderes Vergleichsergebnis produziert wird als über den Umweg über `type_select`. `handle_exceptions` besitzt zwei Spezialisierungen für die Freigabe bzw. Sperrung einer Ausnahme:

```
template <bool> class handle_exceptions;

template <> class handle_exceptions <true> {
    static void reg(const type_info& ti);
    static void unreg(const type_info& ti);
    template <class T,bool,bool> friend class NotifyHandler;
};//end struct

template <> class handle_exceptions<false> {
    static void reg(const type_info& ti);
    static void unreg(const type_info& ti);
    template <class T,bool,bool> friend class NotifyHandler;
};//end struct
```

Die Sperrung oder Freigabe erfolgt durch Notierung des Objekttyps in einer Liste. Ein Objekt von `type_info` kann zwar nicht kopiert werden, jedoch ist die Sicherung einer konstanten Referenz auf ein von `typeid(..)` geliefertes Objekt möglich (*der Operator liefert konstante Referenzen auf persistente Kontrollelemente des Compilerbauers, sodass keine Gefahr der vorzeitigen Zerstörung eines solchen Objektes besteht*). Insgesamt sieht dies folgendermaßen aus:

```
struct exc_entry {
    const type_info& ti;
    int   cnt;

    exc_entry(const type_info& t): ti(t), cnt(0) {}
    exc_entry(const exc_entry& ec):
                        ti(ec.ti), cnt(ec.cnt) {}

    bool operator==(const type_info& t)
           { return t==ti;}
    bool operator==(const exc_entry& ec)
           {return ec.ti==ti;}
private:
    exc_entry();
    exc_entry& operator=(const exc_entry&);
};//end class

typedef list<exc_entry> SpecList;

SpecList& Liste(){
    static SpecList l;
```

```
    return l;
}//end function
```

Bei Eintrag eines Objekttyps wird die Liste zunächst durchsucht, ob der Type bereits notiert ist, und im Negativfall ein neues Listenelement eingefügt. Die Notierung erfolgt mittels des Attributs `cnt`, dessen Wert Freigabe oder Sperrung bedeutet.

```
SpecList::iterator get(const type_info& ti){
    SpecList::iterator it;
    for(it=Liste().begin();it!=Liste().end();++it)
        if(*it==ti)
            return it;
    Liste().push_front(exc_entry(ti));
    return Liste().begin();
}//end function

void handle_exceptions<true>::reg
                        (const type_info& ti){
    get(ti)->cnt++;
}//end function

void handle_exceptions<true>::unreg
                        (const type_info& ti){
    get(ti)->cnt--;
}//end function

void handle_exceptions<false>::reg
                        (const type_info& ti){
    get(ti)->cnt-=100;
}//end function

void handle_exceptions<false>::unreg(const type_info& ti){
    get(ti)->cnt+=100;
}//end function
```

Die Methode `exception_allowed(..)` auf dieser Basis zu konstruieren überlasse ich Ihnen.
Die Ausnahmen werden mit Hilfe eines Objektes der Klasse `Throw` geworfen, die ebenfalls die Strategien (a)–(c) beherrschen muss:

```
template <class T> struct Throw {
    typedef select_type<T,true> d_type;
    typedef select_type<T,false> m_type;
    typedef select_type<extended_exception,false>
                                        a_type;

    Throw(const T& t){
        if(exception_allowed(typeid(d_type)))
            throw T(t);
```

```
    }//end function

    Throw(string s, const T& t){
        if(exception_allowed(typeid(m_type)))
            throw managed_exception<T>(s,t);
    }//end function
};

template <> struct Throw<string> {
    typedef select_type<extended_exception,false>
                                        s_type;

    Throw(string s){
        if(exception_allowed(typeid(s_type)))
            throw extended_exception(s);
    }//end function
};//end struct
```

Je nach verwendetem Konstruktion wird einer der drei Ausnahmetypen geworfen, vorausgesetzt, der Typ ist über ein Notify-Objekt registriert und freigegeben. Fassen wir zusammen: der wieder einmal etwas größere Aufwand bei der Implementation der Steuerklassen zahlt sich durch eine einfache Bedienung aus. Die folgende Programmsequenz erlaubt das Werfen verschiedener Ausnahmen:

```
try{
    NotifyHandler<T> nh1;
    NotifyHandler<T,false> nh2;
    Notifyhandler<extented_exception> nh3;
    ...
    Throw<T>(t);
    Throw<T>("Fehler",t);
    Throw<string>("Fehler");
    ...
}catch(T const & t){
    ...
}catch(managed_exception<T>6 me){
    ...
}catch(extended_exception ex){..}
```

Diese Technik ähnelt sehr stark dem Standard-Ausnahmemanagement und weniger dem Objekt-Leitmechanismus, den wir in der ersten Implementation realisiert haben. Sie gibt uns die Möglichkeit, Ausnahmen gezielt individuell ein- oder auszuschalten, ohne eine Anwendung komplett zu einem bestimmten Verhalten zu zwingen. In Bibliotheken können via

```
bool do_what(..){
    ...
    if(special_case==true){
```

```
        Throw<T>(obj);
        return false;
    }//endif
    ...
    return true;
}//endfunction
```

beide Strategien (*Ausnahme oder Rückgabewert*) nebeneinander realisiert werden, wobei das Weitere dem Nutzer überlassen bleibt.

5.5 Anwendungsbeispiel: Transaktionsmanagement

Wir betrachten zum Abschluss ein etwas einfacheres Anwendungsbeispiel für Ausnahmen, die auf einen Ort beschränkbar sind und mit weniger Aufwand als im letzten Kapitel beschrieben auskommen. Transaktionen sind Vorgänge, die nach dem Schema „alles oder nichts“ abgewickelt werden müssen. Ein etwas gekünsteltes Beispiel haben Sie ja in einer der Aufgaben schon untersucht. Hier stelle ich ein komplettes Anwendungsbeispiel ohne Formulierung von Aufgaben vor. Für Ihre weitere Arbeit sollten Sie anschließend das Ausnahmemanagement betreffend bestens gerüstet sein.

Der Begriff „Transaktion“ ist meist an die Arbeit mit Datenbanken gekoppelt, allerdings ist das kein Muss. Das klassische Beispiel ist die Durchführung einer Banküberweisung, bei der am Ende der Aktion genauso viel Geld auf dem einen Konto abgebucht sein muss, wie einem anderen Konto zufließt. Beteiligt bei Transaktionen sind immer mehrere Objekte, die jeweils eine bestimmte Aktion ausführen müssen. Die Transaktion ist erfolgreich abgeschlossen, sobald alle Objekte den erfolgreichen Vollzug ihrer Ausgabe melden. Gelingt das nicht, meldet also mindestens ein Objekt die Nichtdurchführbarkeit einer Aktion, so müssen alle bereits ausgeführten Aktionen wieder rückgängig gemacht werden. Die Objekte stellen hierfür (*mindestens*) zwei Methoden zur Verfügung:

```
obj_k.proceed(...);// Die Aktion wird durchgeführt.
                   // Hierbei kann es zu
                      // Misserfolgen kommen
obj_k.roll_back(..);// Die letzte (erfolgreiche)
                      // Aktion wird wieder rückgängig
                      // gemacht
obj_k.commit()        // optional, siehe Text
```

Der Einfachheit halber wollen wir annehmen, dass die `roll_back()`–Funktionen ohne Argument auskommen. Meist wird noch eine dritte Methode (`commit()`) zur Verfügung gestellt, die das Ergebnis fixiert, das heißt bei einem versehentlichen Aufruf von `roll_back()` wird die Transaktion nicht mehr rückgängig gemacht.

Die theoretische Vorgehensweise ist denkbar einfach und erfordert pro Vorgang (*bis auf den letzten*) zwei zusätzliche Anweisungen:

(a) Der Vorgang wird ausgeführt. Tritt hierbei eine Ausnahme auf, so ist der Datenzustand definitionsgemäß der gleiche wie zu Beginn der Aktion, so dass nichts weiter unternommen werden muss.
(b) Nach Beenden des Vorgangs werden eine Objekt- und eine Methodenreferenz (*letztere für die* `roll_back()` *–Methode*) in eine Kellerliste geschrieben. Anschließend wird mit dem nächsten Vorgang fortgefahren.
(c) Sind alle Vorgänge bearbeitet, wird die Kellerliste komplett gelöscht. Die Transaktion ist erfolgreich beendet.
(d) Tritt eine Ausnahme auf, so wird die Anweisung, die zur Löschung der Kellerliste führt, nicht mehr ausgeführt. In der Liste sind alle Vorgänge vorhanden, die rückgängig gemacht werden müssen. Diese werden abgearbeitet. Die Transaktion ist anschließend mit dem Ausgangszustand erfolglos beendet.

Für die Verwaltung der Kellerliste verwenden wir den gleichen Trick wie zuvor: Wir deklarieren statische Variable, die bei Verlassen der Funktion zerstört werden. Die folgende Implementation enthält einige Programmiertechniken, die die Anwendung erleichtern und Fehlbedienungen verhindern sollen. Das angestrebte einheitliche und sehr einfache Aufrufschema mit den Ausnahme–Kontrollobjekten der (*Wächter-*)Klasse `Guard` sieht folgendermaßen aus:

```
Class_k obj_k;
FILE * f;
...
try{
    obj_k.proceed();
    Guard gk = MakeGuard(obj_k,Class_k::roll_back);
    ...
    f = fopen(...);
    Guard gf = MakeGuard(f,fclose);
    ...
    gk.Dismiss();
    gf.Dismiss();
} catch(...) {}
```

Im Unterschied zu der Implementation im letzten Kapitel betrachten wir als kontrollierte Objekte gewöhnliche Variable, deren Gültigkeitsbereich größer als der `try-catch`–Block ist. Die Kontrollobjekte selbst werden nur im `try`-Bereich angelegt. Bereits das vorzeitige Verlassen dieses Blocks reicht aus, um die Umkehraktionen einzuleiten, ohne dass im `catch`–Block eine Variable oder weitere Anweisungen notwendig wären (*der Fangblock fängt alle Ausnahmen, die bis hier durchkommen, das heißt ... ist das Argument von* `catch`*und kein Platzhalter für irgendeine Variable*). Die zusammengehörenden `try-catch`–Blöcke befinden sich

in der gleichen Methode, das heißt eine Methoden übergreifende Ausnahmeabwicklung ist hier nicht vorgesehen. Bei Abbau der Objekte sollen folgende Aktionen durchgeführt werden, sofern die Methode `Dismiss()` nicht erreicht wird:

```
gk:  if(!dismissed)
         obj_k.roll_back();

gf:  if(!dismissed)
         fclose(f);
```

Die Funktion der Methode `Dismiss()` dürfte damit bereits geklärt sein: Ein Attribut `dismissed` wird zu Beginn auf `false` und bei Aufruf von `Dismissed()` auf `true` gesetzt.

Wir konstruieren nun die Kontrollklasse in einer Form, die den Compiler eine Reihe von Programmierfehlern erkennen lässt und außerdem das bereits eingangs angesprochene Problem von Ausnahmen in Destruktoren beleuchtet. Auch hier sehen wir die Möglichkeit mehrerer kontrollierter Objekte vor, die wir jeweils in Form einer eigenen Klasse implementieren, die von einer gemeinsamen Basisklasse erbt. Die beiden oben dargestellten Beispielvariablen `gk` und `gf` sind Instanzen solcher verschiedener Klassen. Wir beginnen mit der Basisklasse, die für die Abwicklung von Destruktoren eingerichtet ist:

```
class GuardBase {
public:
    void Dismiss() const {dismissed=true;};
protected:
    GuardBase() {dismissed=false;}
    GuardBase(const GuardBase& g){
        dismissed=g.dismissed;
        g.Dismiss();
    };
    ~GuardBase() {};

    template <class T> static
    void SafeExecute(T& t){
        if (!t.dismissed_)
            try    { j.Execute();}
            catch(...){}
    };//end function

    mutable bool dismissed;

private:
    GuardBase& operator=(const GuardBase&){};
};//end class
```

Lediglich die Methode `Dismiss()` ist öffentlich. Alle anderen Methoden werden von der erbenden Klasse bedient. Zuweisungen zwischen Guard–Objekten schließen wir dadurch aus, dass der Zuweisungsoperator vom Typ `private` ist, also auch

von den erbenden Klassen nicht verwendet werden kann. Es kann also jeweils nur ein Kontrollobjekt erzeugt werden, das für den deklarierenden Block gilt und seine Verantwortung nicht übertragen kann.[11] Von dieser Klasse lassen wir nun spezielle Anwendungsklassen erben, beispielsweise hier die Klasse der Variablen `gk`:

```
template <class T, typename Func> class GuardImpl :
    public GuardBase {
public:
    GuardImpl(T& t, Func f): obj(t), func(f) {};

    ~GuardImpl(){
        SafeExecute(*this);
    };//end function

    void Execute(){
        (obj_.*func)();
    };//end function

protected:
    T& obj_;
    Func func;
};//end class
```

Das Erzeugen eines Guard–Objektes ist stets mit der Übergabe eines zu kontrollierenden Objektes verknüpft, auf das lediglich eine Referenz gespeichert wird.[12] Beachten Sie, dass dieses Modell völlig anders ist als das von uns im letzten Teilkapitel entwickelte! Dort hatten wir es mit Zeigerobjekten mit einem beliebigen Lebenszyklus zu tun, die einer bestimmten Vererbungshierarchie gehorchten, hier müssen die Objekte länger leben als die Kontrollobjekte, müssen aber keiner Vererbungshierarchie gehorchen, sondern werden als `template` –Parameter übergeben.

Aufwendig ist lediglich der Destruktorablauf gestaltet: grundsätzlich ist ja nicht auszuschließen, dass auch die Korrekturfunktion eine Ausnahme wirft.[13] Innerhalb eines Destruktors wäre dies allerdings fatal, da ein `try-catch`–Formalismus nicht zulässig ist (*ein Objekt wäre gegebenenfalls „ halb“ abgebaut, wenn das System infolge des Ausnahmewurfes erneut den Destruktor aufruft. An was sollte es sich dann halten?*). Hier geht es deshalb ein wenig hin und her und hier ist auch der Grund dafür zu suchen, dass eine Vererbungshierarchie aufgebaut wird (*haben Sie*

---

[11] Das hat gewissen Konsequenzen, wie sich noch zeigen wird.

[12] Ein Kopierkonstruktor ist für die Basisklasse zwar definiert und gibt die Verantwortung korrekt weiter, wird aber im allgemeinen nicht benötigt.

[13] Eine solche Ausnahme hat nichts mit der Transaktion zu tun, denn sonst würden wir sie unterdrücken. Hier berücksichtigen wir lediglich die Möglichkeit, dass im Destruktor Methoden aufgerufen werden, die ein methodenübergreifendes Ausnahmemanagement aus ganz anderen Gründen implementiert haben und deren Auswirkungen wir in diesem speziellen Fall unterdrücken müssen.

sich nicht schon gefragt, warum das Klassenmodell so kompliziert ist?), jedoch ist der Ablauf nun ausführungssicher:

- DerDestruktor `~GuardImpl` in der Oberklasse ruft die statische Methode `SafeExecute(..)` in der Basisklasse auf. Diese beinhaltet die sichere Ablaufsteuerung in Form eines `try-catch`–Blockes mit unbedingtem Fangen jeder Ausnahme, ist aber nicht direkt in der erbenden Klasse implementiert, um (*fehleranfällige*) Kodewiederholungen bei der Implementation weiterer Klassen zu vermeiden. Da die Methode statisch ist, findet die Ausnahmebehandlung nicht im Destruktorkontext statt, sondern in einer unteren Ebene, kann also auch nicht zu Konflikten durch Mehrfachaufrufe führen. Ob Ausnahme oder nicht: Der Destruktor kann anschließend korrekt zu Ende durchgeführt werden.
- Die Methode `GuardBase::SafeExecute(..)` wird in der Basisklasse definiert. Ihr Parameter ist das Objekt der erbenden Klasse, welches die Ausführungsfunktion `Execute()` aufruft. Diese kann erst in der erbenden Klasse implementiert werden, da nun die Objektfunktion aufgerufen werden muss, von der die Basisklasse natürlich noch nichts weiß. `SafeExecute(..)` muss deshalb als Vorlagenfunktion deklariert werden und die erbende Klasse als `template`–Parameter erhalten, um überhaupt arbeitsfähig zu sein.
- Die Ausführungsmethode des kontrollierten Objektes befindet sich nun wieder in der erbenden Klasse und wird in der Methode `Execute()` aufgerufen. Eventuell hier auftretende Ausnahmen werden in `SafeExecute` wie beschrieben gefangen.
- Abschließend wird der Destruktor der Basisklasse als leerer Destruktor ausgeführt.

Betrachten wir nun noch einmal unser Aufrufschema. Dort haben wir die Klassenbezeichnung `Guard` benutzt. Scheinbar handelt es sich um eine Variable und keine Zeigervariable, und dieser Variablen wird über ein Funktionsaufruf ein Wert zugewiesen. Dabei muss aber gelten:

- Die Variable muss trotz ihres einheitlichen Namens für alle Anwendungen die beschriebene Vererbung unterstützen.
- Der Zuweisungsoperator ist im privaten Bereich geschützt und kann nicht verwendet werden.

Damit kommt für die Guard–Deklaration nur eine Referenz auf die Basisklasse in Frage, die wir durch

```
typedef const GuardBase& Guard;
```

definieren. An die (*konstante*) Referenz wird durch die Deklarationsanweisung ein (*temporäres*) Objekt gebunden. Dieses bleibt nun innerhalb des Geltungsbereiches der Referenz erhalten und wird erst bei Verlassen des Bereiches zerstört. Da nach C++–Sprachstandard die `const`–Vereinbarung Voraussetzung für diese Technik ist, haben wir das Attribut `dismissed` als `mutual` definieren müssen (*s.o.*), um in

der Basisklasse alle Methoden als `const` definieren zu können.[14] Der beschriebene Destruktorablauf sorgt überdies zusammen mit der Referenz dafür, dass aufgerufene Methoden auch ohne `virtual`–Vereinbarungen korrekt aufgelöst werden können.

Das temporäre Objekt erzeugen wir schließlich mit der Funktion

```
template <class T, typename Func> GuardImpl<T,Func>
MakeGuard(T& t, Func f){
    return GuardImpl <T,Func>(t,f);
};//end function
```

Zum Abschluss hier noch ein vollständiges Aufrufbeispiel:

```
class MyClass {
public:
    void doSomething(){...};
    ...
};//end MyClass
int main(){
    MyClass obj;
    ...
    try{
      Guard g=MakeGuard(obj,&MyClass::doSomething);
        ...
    }catch(...){
    }
```

Bei konstantem Anwendungsaufruf durch Definition anderer Klassen des Typs `GuardImpl` und Schreiben anderer Versionen der Funktion `MakeGuard(..)` sind weitere Anwendungsfälle leicht abdeckbar, auch mit beliebigen Argumenten als Funktionsparameter, die in den Vorlagendefinitionen als eigene Typnamen auftreten müssen.

Aufgabe. Implementieren Sie eine Version, die eine Ausnahmebehandlungsfunktion mit einem Argument abwickelt.

[14] Die Beschränkung auf const ist leicht einzusehen. Für den Compiler ist es nämlich bei Zuweisungen aus anderen Modulen nicht kontrollierbar, ob sich hinter einer Referenz etwas konstantes verbirgt oder nicht. Da er aber im Standardfall die Funktion garantieren muss, bleibt nur die const–Deklaration. Will der Programmieren etwas anderes, so muss er dies (*wie beispielsweise beim* const_cast) dem Compiler separat mitteilen und übernimmt damit auch die Verantwortung für sein Tun.