
Freitag
Samstag
Sonntag

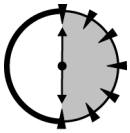
Sonntagmorgen

Teil 5

Vererbung

Checkliste

- ☑ Vererbung definieren
- ☑ Von einer Basisklasse erben
- ☑ Die Basisklasse konstruieren
- ☑ die Beziehungen IS_A und HAS_A vergleichen



30 Min.

In dieser Sitzung diskutieren wir Vererbung. *Vererbung* ist die Fähigkeit einer Klasse, auf Fähigkeiten oder Eigenschaften einer anderen Klasse zurückzugreifen. Ich z.B. bin ein Mensch. Ich erbe von der Klasse *Mensch* bestimmte Eigenschaften, wie z.B. meine Fähigkeit zu (mehr oder weniger) intelligenter Konversation, und meine Abhängigkeit von Luft, Wasser und Nahrung. Diese Eigenschaften sind nicht einzigartig für Menschen. Die Klasse *Mensch* erbt diese Abhängigkeit von Luft, Wasser und Nahrung von der Klasse *Säugetier*.

21.1 Vorteile von Vererbung

Die Fähigkeit, Eigenschaften nach unten weiterzugeben, ist ein mächtige. Sie erlaubt es uns, Dinge auf ökonomische Art und Weise zu beschreiben. Wenn z.B. mein Sohn fragt »Was ist eine Ente?« kann ich sagen »Es ist ein Vogel, der Quakquak macht.« Was immer Sie über diese Antwort denken mögen, übermittelt sie ihm einiges an Informationen. Er weiß, was ein Vogel ist, und jetzt weiß er all diese Dinge für Enten, plus die zusätzliche Quakquak-Eigenschaft.

Es gibt mehrere Gründe, weshalb Vererbung in C++ eingeführt wurde. Sicherlich ist der wichtigste Grund die Möglichkeit, Vererbungsbeziehungen auszudrücken. (Ich werde darauf gleich zurückkommen.) Ein weniger wichtiger Grund ist der, den Schreibaufwand zu reduzieren. Nehmen Sie an, Sie haben eine Klasse `Student`, und wir sollen eine neue Klasse `GraduateStudent` hinzufügen. Vererbung kann die Anzahl der Dinge, die wir in eine solche Klasse packen müssen, drastisch reduzieren. Alles, was wir in der Klasse `GraduateStudent` wirklich brauchen, sind die Dinge, die den Unterschied zwischen Studenten und graduierten Studenten beschreiben.

Wichtiger ist das verwandte Reizwort der 90-er Jahre, Wiederverwendung. Softwarewissenschaftler haben vor einer gewissen Zeit festgestellt, dass es keinen Sinn macht, in jedem Softwareprojekt bei Null zu beginnen, und die gleichen Softwarekomponenten immer wieder neu zu schreiben.

Vergleichen Sie diese Situation bei der Software mit anderen Industrien. Wie viele Autohersteller fangen bei jedem Auto ganz von vorne an? Und selbst wenn sie das täten, wie viele würden beim nächsten Modell wieder ganz von vorne beginnen? Praktiker in anderen Industrien haben es sinnvoller gefunden, bei Schrauben, Muttern und auch größeren Komponenten wie Motoren und Kompressoren zu beginnen.

Unglücklicherweise ist, mit Ausnahme der sehr kleinen Funktionen in der Standardbibliothek von C, nur sehr wenig Wiederverwendung von Softwarekomponenten zu sehen. Ein Problem ist, dass es fast unmöglich ist, eine Komponente in einem früheren Programm zu finden, die exakt das tut, was Sie brauchen. Im Allgemeinen müssen diese Komponenten angepasst werden.

Es gibt eine Daumenregel die besagt »Wenn Sie es geöffnet haben, haben Sie es zerbrochen«. Mit anderen Worten, wenn Sie eine Funktion oder Klasse modifizieren müssen, um sie an Ihre Anwendung anzupassen, müssen Sie wieder alles neu testen, nicht nur die Teile, die Sie hinzugefügt haben. Änderungen können irgendwo im Code Bugs verursachen. (»Wer den Code zuletzt angefasst hat, muss den Bug fixen«.)

Vererbung ermöglicht es, bestehende Klassen an neue Anwendungen anzupassen ohne sie verändern zu müssen. Von der bestehenden Klasse wird eine neue Unterklasse abgeleitet, die alle nötigen Zusätze und Änderungen enthält.

Das bringt einen dritten Vorteil. Nehmen Sie an, wir erben von einer existierenden Klasse. Später finden wir heraus, dass die Basisklasse einen Fehler enthält und korrigiert werden muss. Wenn wir die Klasse zur Wiederverwendung modifiziert haben, müssen wir in jeder Anwendung einzeln auf den Fehler testen und ihn korrigieren. Wenn wir von der Klasse ohne Änderungen geerbt haben, können wir die berichtigte Klasse sicher ohne Weiteres übernehmen.

21.2 Faktorieren von Klassen

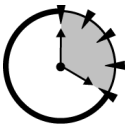
Um unsere Umgebung zu verstehen, haben die Menschen umfangreiche Begrifflichkeiten eingeführt. Unser *Fido* ist ein Spezialfall von *Rüde*, was ein Spezialfall von *Hund* ist, was ein Spezialfall von *Säugetier* ist usw. Das formt unser Verständnis unserer Welt.

Um ein anderes Beispiel zu gebrauchen, ist ein Student ein spezieller Typ Person. Wenn ich das gesagt habe, weiß ich bereits viele Dinge über Studenten. Ich weiß, dass Sie eine Sozialversicherungsnummer haben, dass Sie zu viel fernsehen, dass sie zu schnell fahren, und nicht genug üben. Ich weiß all diese Dinge, weil es Eigenschaften aller Leute sind.

In C++ bezeichnen wir das als *Vererbung*. Wir sagen, dass die Klasse `Student` von der Klasse `Person` *erbt*. Wir sagen auch, dass die Klasse `Person` die *Basisklasse* von `Student` ist und `Student` eine *Unterklasse* von `Person` ist. Schließlich sagen wir, dass ein `Student` IS_A `Person` (ich verwende die Großbuchstaben als meine Art, um diese eindeutige Beziehung zu bezeichnen). C++ teilt diese Terminologie mit anderen objektorientierten Sprachen.

Beachten Sie, dass obwohl `Student` IS_A `Person` wahr ist, das Gegenteil nicht der Fall ist. (Eine Aussage wie diese bezieht sich immer auf den allgemeinen Fall. Es kann sein, dass eine bestimmte `Person` ein `Student` ist.) Eine Menge Leute, die zur Klasse `Person` gehören, gehören nicht zur Klasse `Student`. Das liegt daran, dass die Klasse `Student` Eigenschaften besitzt, die sie mit der Klasse `Person` nicht teilt. Z.B. hat `Student` einen mittleren Grad, aber `Person` hat das nicht.

Die Vererbungsbeziehung ist jedoch transitiv. Wenn ich z.B. eine neue Klasse `GraduateStudent` als Unterklasse von `Student` einführe, muss `GraduateStudent` auch `Person` sein. Es muss so aussehen: wenn `GraduateStudent` IS_A `Student` und `Student` IS_A `Person`, dann `GraduateStudent` IS_A `Person`.



20 Min.

21.3 Implementierung von Vererbung in C++

Um zu demonstrieren, wie Vererbung in C++ ausgedrückt wird, lassen Sie uns zu dem Beispiel `GraduateStudent` zurückkehren und dieses mit einigen exemplarischen Elementen ausstatten:

```
// GSInherit - demonstriert, wie GraduateStudent von
//             Student die Eigenschaften eines
//             Studenten erben kann
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Advisor - nur eine Beispielklasse
class Advisor
{
};

// Student - alle Informationen über Studenten
class Student
{
  public:
    Student()
    {
        // initialer Zustand
        pszName = 0;
        nSemesterHours = 0;
        dAverage = 0;
    }
    ~Student()
    {
        // wenn es einen Namen gibt ...
        if (pszName != 0)
        {
            // ... dann gib den Puffer zurück
            delete pszName;
            pszName = 0;
        }
    }

    // addCourse - fügt den Effekt eines absolvierten
    //             Kurses mit dGrade zu dAverage
    //             hinzu
    void addCourse(int nHours, double dGrade)
    {
        // aktuellen gewichteten Mittelwert
```

```
        int ndGradeHours = (int)(nSemesterHours * dAverage + dGrade);

        // beziehe die absolvierten Stunden ein
        nSemesterHours += nHours;

        // berechne neuen Mittelwert
        dAverage = ndGradeHours / nSemesterHours;
    }

    // die folgenden Zugriffsfunktionen geben
    // der Anwendung Zugriff auf wichtige
    // Eigenschaften
    int hours( )
    {
        return nSemesterHours;
    }
    double average( )
    {
        return dAverage;
    }

  protected:
    char*  pszName;
    int    nSemesterHours;
    double dAverage;

    // Kopierkonstruktor - ich will nicht, dass
    //                     Kopien erzeugt werden
    Student(Student& s)
    {
    }
};

// GraduateStudent - diese Klasse ist auf die
//                   Studenten beschränkt, die ihr
//                   Vordiplom haben
class GraduateStudent : public Student
{
  public:
    GraduateStudent()
    {
        dQualifierGrade = 2.0;
    }

    double qualifier( )
    {
        return dQualifierGrade;
    }

  protected:
    // alle graduierten Studenten haben einen Advisor
    Advisor advisor;

    // das ist der Grad, unter dem ein
    // GraduateStudent den Kurs nicht
    // erfolgreich absolviert hat
```

```
    double dQualifierGrade;
};

int main(int nArgc, char* pszArgs[])
{
    // erzeuge einen Studenten
    Student llu;

    // und jetzt einen graduierten Studenten
    GraduateStudent gs;

    // das Folgende ist völlig korrekt
    llu.addCourse(3, 2.5);
    gs.addCourse(3, 3.0);

    // das Folgende aber nicht
    gs.qualifier();      // das ist gültig
    llu.qualifier();     // das aber nicht
    return 0;
}
```

Die Klasse `Student` wurde in der gewohnten Weise deklariert. Die Deklaration der Klasse `GraduateStudent` unterscheidet sich davon. Der Name der Klasse, gefolgt von dem Doppelpunkt, gefolgt von public `Student` deklariert die Klasse `GraduateStudent` als Unterklasse von `Student`.

Das Schlüsselwortes `public` impliziert, dass es sicherlich auch eine protected-Vererbung gibt. Das ist der Fall, ich möchte jedoch diesen Typ Vererbung für den Moment aus der Diskussion heraus lassen.

Die Funktion `main( )` deklariert zwei Objekte, `llu` und `gs`. Das Objekt `llu` ist ein konventionelles `Student`-Objekt, aber das Objekt `gs` ist etwas Neues. Als ein Mitglied einer Unterklasse von `Student`, kann `gs` alles tun, was `llu` tun kann. Es hat die Datenelemente `pszName`, `nSemesterHours` und `dAverage` und die Elementfunktion `addCourse( )`. Buchstäblich gilt, `gs IS_A Student` – `gs` ist nur ein wenig mehr als ein `Student`. (Sie werden es am Ende des Buches sicher nicht mehr ertragen können, dass ich »`IS_A`« so oft benutze.) In der Tat hat die Klasse `GraduateStudent` die Eigenschaft `qualifier( )`, die `Student` nicht besitzt.

Die nächsten beiden Zeilen fügen den beiden Studenten `llu` und `gs` einen Kurs hinzu. Erinnern Sie sich daran, dass `gs` auch ein `Student` ist.

Eine der letzten Zeilen in `main( )` ist nicht korrekt. Es ist in Ordnung die Methode `qualifier( )` für das Objekt `gs` aufzurufen. Es ist nicht in Ordnung, die Eigenschaft `qualifier` für das Objekt `llu` zu verwenden. Das Objekt `llu` ist nur ein `Student` und hat nicht die Eigenschaften, die für `GraduateStudent` einzigartig sind.

Betrachten Sie das folgende Szenario:

```
// fn - führt eine Operation auf Student aus
void fn(Student &s)
{
    // was immer fn tun möchte
}

int main(int nArgc, char* pszArgs[])
{
    // erzeuge einen graduierten Studenten ...
    GraduateStudent gs;

    // ... übergib ihn als einfachen Studenten
    fn(gs);
    return 0;
}
```

Beachten Sie, dass die Funktion `fn( )` ein Objekt vom Typ `Student` als Argument erwartet. Der Aufruf von `main( )` übergibt der Funktion ein Objekt aus der Klasse `GraduateStudent`. Das ist in Ordnung, weil (um es noch einmal zu wiederholen) »ein `GraduateStudent` IS_A `Student`.«

Im Wesentlichen entstehen die gleichen Bedingungen, wenn eine Elementfunktion von `Student` mit einem `GraduateStudent`-Objekt aufgerufen wird. Z.B.:

```
int main(int nArgc, char* pszArgs[])
{
    GraduateStudent gs;
    gs.addCourse(3, 2.5);//ruft Student::addCourse( )
    return 0;
}
```

21.4 Unterklassen konstruieren

Obwohl eine Unterklasse Zugriff hat auf die protected-Elemente der Basisklasse und diese in ihrem eigenen Konstruktor initialisieren kann, möchten wir gerne, dass sich die Basisklasse selber konstruiert. Das ist in der Tat, was passiert. Bevor die Kontrolle über die öffnende Klammer des Konstruktors von hinwegkommt, geht sie zuerst auf die Defaultkonstruktor von `Student` über (weil kein anderer Konstruktor angegeben wurde). Wenn `Student` auf einer weiteren Klasse basieren würde, wie z.B. `Person`, würde der Konstruktor dieser Klasse aufgerufen, bevor der Konstruktor von `Student` die Kontrolle bekommt. Wie ein Wolkenkratzer wird ein Objekt von seinem Fundament die Klassenstruktur aufwärts aufgebaut.

Wie mit Elementobjekten, ist es manchmal nötig, Argumente an den Konstruktor der Basisklasse zu übergeben. Wir tun dies in fast der gleichen Weise, wie bei den Elementobjekten, wie das folgende Beispiel zeigt:

```
// Student - diese Klasse enthält alle Typen
//          von Studenten
class Student
{
  public:
    // Konstruktor - definiere Defaultargument,
```

```
    //              um auch einen Defaultkonstruktor
    //              zu haben
    Student(char* pszName = 0)
    {
        // initialer Zustand
        this->pszName = 0;
        nSemesterHours = 0;
        dAverage = 0.0;

        // wenn es einen Namen gibt ...
        if (pszName != 0)
        {
            this->pszName =
                new char[strlen(pszName) + 1];
            strcpy(this->pszName, pszName);
        }
    }
    ~Student()
    {
        // wenn es einen Namen gibt ...
        if (pszName != 0)
        {
            // ... dann gib den Puffer zurück
            delete pszName;
            pszName = 0;
        }
    }
    // ... Rest der Klassendefinition ...
};

// GraduateStudent - diese Klasse ist auf die
//                   Studenten beschränkt, die ihr
//                   Vordiplom haben
class GraduateStudent : public Student
{
  public:
    // Konstruktor - erzeuge graduierten Studenten
    //               mit einem Advisor, einem Namen
    //               und einem Qualifizierungsgrad
    GraduateStudent(
            Advisor &adv,
            char*   pszName = 0,
            double  dQualifierGrade = 0.0)
                : Student(pName),
                  advisor(adv)
    {
        // wird erst ausgeführt, nachdem die anderen
        // Konstruktoren aufgerufen wurden
        dQualifierGrade = 0;
    }
  protected:
    // alle graduierten Studenten haben einen Advisor
    Advisor advisor;

    // das ist der Grad, unter dem ein
    // GraduateStudent den Kurs nicht
```

```
    // erfolgreich absolviert hat
    double dQualifierGrade;
};
void fn(Advisor &advisor)
{
    // graduierten Studenten erzeugen
    GraduateStudent gs(»Marion Haste«,
                       advisor,
                       2.0);
    //... was immer diese Funktion tut ...
}
```

Hier wird ein `GraduateStudent`-Objekt mit einem Advisor erzeugt, dessen Name »Marion Haste« und dessen Grad gleich 2.0 ist. Der Konstruktor von `GraduateStudent` ruft den Konstruktor `Student` auf, und übergibt den Namen des Studenten. Die Basisklasse wird konstruiert vor allen anderen Elementobjekten; somit wird der Konstruktor von `Student` vor den Konstruktor von `Advisor` aufgerufen. Nachdem die Basisklasse konstruiert wurde, wird das `Advisor`-Objekt `advisor` mittels Kopierkonstruktor konstruiert. Erst dann kommt der Konstruktor von `GraduateStudent` zum Zuge.

Die Tatsache, dass die Basisklasse zuerst erzeugt wird, hat nichts mit der Ordnung der Konstruktoranweisungen hinter dem Doppelpunkt zu tun. Die Basisklasse wäre auch dann vor den Datenelementen konstruiert worden, wenn die Anweisungen `advisor(adv), Student(pszName)` ***gelautet hätten. Es ist jedenfalls eine gute Idee, diese Klauseln in der Reihenfolge zu schreiben, in der sie ausgeführt werden, nur um niemanden zu verwirren.***

10 Min.

Gemäß unserer Regel, dass Destruktoren in der umgekehrten Reihenfolge aufgerufen werden wie die Konstruktoren, bekommt der Destruktor von `GraduateStudent` zuerst die Kontrolle. Nachdem er seine letzten Dienste erbracht hat, geht die Kontrolle auf den Destruktor von `Advisor` und dann auf den Destruktor von `Student` über. Wenn `Student` von einer Klasse Person abgeleitet wäre, ginge die Kontrolle auf den Destruktor von `Person` nach `Student` über.

Der Destruktor der Basisklasse `Student` ***wird ausgeführt, obwohl es keinen expliziten Destruktor*** `~GraduateStudent` ***gibt.***

Das ist logisch. Der wenige Speicher, der schließlich zu einem `GraduateStudent`-Objekt wird, wird erst in ein `Student`-Objekt konvertiert. Dann ist es die Aufgabe des Konstruktors `GraduateStudent`, seine Transformation in ein `GraduateStudent`-Objekt zu vervollständigen. Der Destruktor kehrt diesen Prozess einfach um.

Beachten Sie einige wenige Dinge in diesem Beispiel. Erstens wurden Defaultargumente im Konstruktor `GraduateStudent` ***bereitgestellt, um diese Fähigkeit an die Basisklasse*** `Student` ***weiterzugeben. Zweitens können Defaultwerte für Argumente nur von rechts nach links angegeben werden. Das Folgende ist nicht möglich:***
`GraduateStudent(char* pszName = 0, Advisor& adv) ...`
Die Argumente ohne Defaultwerte müssen zuerst kommen.

Beachten Sie, dass die Klasse `GraduateStudent` ein `Advisor`-Objekt in der Klasse enthält. Es enthält keinen Zeiger auf ein `Advisor`-Objekt. Letzteres würde so geschrieben werden:

```
class GraduateStudent : public Student
{
  public:
    GraduateStudent(
        Advisor& adv,
        char*    pszName = 0)
        : Student(pName),
    {
        pAdvisor = new Advisor(adv);
    }
  protected:
    Advisor* pAdvisor;
};
```

Hierbei wird die Basisklasse `Student` zuerst erzeugt (wie immer). Der Zeiger wird innerhalb des Body des Konstruktors `GraduateStudent` initialisiert.

21.5 Die Beziehung HAS_A

Beachten Sie, dass die Klasse `GraduateStudent` die Elemente der Klasse `Student` und `Advisor` einschließt, aber auf verschiedene Weisen. Durch die Definition eines Datenelementes aus der Klasse `Advisor` wissen wir, dass ein `GraduateStudent` alle Datenelemente von `Advisor` in sich enthält, und wir drücken das aus, indem wir sagen, `GraduateStudent` HAS_A `Advisor`. Was ist der Unterschied zwischen dieser Beziehung und Vererbung?

Lassen Sie uns ein Auto als Beispiel nehmen. Wir könnten logisch ein Auto als Unterklasse von *Fahrzeug* definieren und dadurch allgemeine Eigenschaften von Fahrzeugen erben. Gleichzeitig hat ein Auto auch einen Motor. Wenn Sie ein Auto kaufen, können Sie logisch davon ausgehen, dass Sie auch einen Motor kaufen.

Wenn nun einige Freunde am Wochenende eine Rallye mit dem Fahrzeug der eigenen Wahl veranstalten, wird sich niemand darüber beschweren, wenn Sie mit ihrem Auto kommen, weil Auto IS_A Fahrzeug. Wenn Sie aber zu Fuß kommen und Ihren Motor unter dem Arm tragen, haben sie allen Grund, erstaunt zu sein, weil ein Motor kein Fahrzeug ist. Ihm fehlen einige wesentliche Eigenschaften, die alle Fahrzeuge haben. Es fehlen dem Motor sogar Eigenschaften, die alle Autos haben.

Vom Standpunkt der Programmierung aus ist es ebenso einfach. Betrachten sie das Folgende:

```
class Vehicle
{
};
class Motor
{
};
class Car : public Vehicle
{
  public:
    Motor motor;
};
void VehicleFn(Vehicle &v);
void motorFn(Motor &m);
int main(int nArgc, char* pszArgs[])
{
   Car c;
   vehicleFn(c);     // das ist erlaubt
   motorFn(c);       // das ist nicht erlaubt
   motorFn(c.motor); // das jedoch schon
   return 0;
}
```

0 Min.

Der Aufruf `vehicleFn(c)` ist erlaubt, weil `c` IS_A `Vehicle`. Der Aufruf `motorFn(c)` ist nicht erlaubt, weil `c` kein Motor ist, obwohl es einen Motor enthält. Wenn beabsichtigt ist, den Teil Motor von `c` an eine Funktion zu übergeben, muss dies explizit ausgedrückt werden, wie im Aufruf `motorFn(c.motor)`.

Natürlich ist der Aufruf `motorFn(c.motor)` ***nur dann erlaubt, wenn*** `c.motor` ***public ist.***

Ein weiterer Unterschied: Die Klasse `Car` hat Zugriff auf die protected-Elemente von `Vehicle`, aber nicht auf die protected-Elemente von `Motor`.

Zusammenfassung

Das Verständnis von Vererbung ist wesentlich für das Gesamtverständnis der objektorientierten Programmierung. Es wird auch benötigt, um das nächste Kapitel verstehen zu können. Wenn Sie den Eindruck haben, dass sie es verstanden haben, gehen Sie weiter zu Kapitel 22. Wenn nicht, lesen Sie dieses Kapitel erneut.

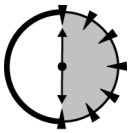
Selbsttest

1. Was ist die Beziehung zwischen einem graduierten Studenten und einem Studenten. Ist es eine Beziehung der Form IS_A oder HAS_A? (Siehe »Die Beziehung HAS_A«)
2. Nennen Sie drei Vorteile davon, dass Vererbung in der Programmiersprache C++ vorhanden ist. (Siehe »Vorteile von Vererbung«)
3. Welcher der folgenden Begriffe passt nicht? *Erbt, Unterklasse, Datenelement* und *IS_A*? (Siehe »Faktorieren von Klassen«)

Polymorphie

Checkliste

- ☑ Elementfunktionen in Unterklassen überschreiben
- ☑ Polymorphie anwenden (alias späte Bindung)
- ☑ Polymorphie mit früher Bindung vergleichen
- ☑ Polymorphie speziell betrachten



30 Min.

Vererbung gibt uns die Möglichkeit, eine Klasse mit Hilfe einer anderen Klasse zu beschreiben. Genauso wichtig ist, dass dadurch die Beziehung zwischen den Klassen deutlich wird. Nochmals, eine Mikrowelle ist ein Typ Ofen. Es fehlt jedoch noch ein Teil im Puzzle.

Sie haben das bestimmt bereits bemerkt, aber eine Mikrowelle und ein herkömmlicher Ofen sehen sich nicht besonders ähnlich. Die beiden Ofentypen arbeiten auch nicht gleich. Trotzdem möchte ich mir keine Gedanken darüber machen, wie jeder einzelne Ofen das »Kochen« ausführt. Diese Sitzung beschreibt, wie C++ dieses Problem behandelt.

22.1 Elementfunktionen überschreiben

Es war immer möglich, eine Elementfunktion in einer Klasse mit einer Elementfunktion in der gleichen Klasse zu überschreiben, solange die Argumente verschieden sind. Es ist auch möglich, ein Element einer Klasse mit einer Elementfunktion einer anderen Klasse zu überschreiben, selbst wenn die Argumente gleich sind.

Vererbung liefert eine weitere Möglichkeit: Eine Elementfunktion in einer Unterklasse kann eine Elementfunktion der Basisklasse überladen.

Betrachten Sie z.B. das einfache Programm `EarlyBinding` in Listing 22-1.

Listing 22-1: Beispielprogramm EarlyBinding

```
// EarlyBinding - Aufrufe von überschriebenen
//                Elementfunktionen werden anhand
//                des Objekttyps aufgelöst
#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    // berechnet das Schulgeld
    double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
  public:
    double calcTuition()
    {
        return 1;
    }
};

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // Student::calcTuition();
    Student s;
    cout << »Der Wert von s.calcTuition ist »
         << s.calcTuition()
         << »\n«;

    // das ruft GraduateStudent::calcTuition();
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition ist »
         << gs.calcTuition()
         << »\n«;
    return 0;
}
```

Ausgabe

```
Der Wert von s.calcTuition ist 0
Der Wert von gs.calcTuition ist 1
```

Wie bei jedem anderen Fall von Überschreiben, muss C++ entscheiden, welche Funktion `calcTuition( )` gemeint ist, wenn der Programmierer `calcTuition( )` aufruft. Normalerweise reicht die Klasse aus, um den Aufruf aufzulösen, und es ist bei diesem Beispiel nicht anders. Der Aufruf `s.calcTuition( )` bezieht sich auf `Student::calcTuition( )`, weil `s` als `Student` deklariert ist, wobei `gs.calcTuition( )` sich auf `GraduateStudent::calcTuition( )` bezieht.

Die Ausgabe des Programms `EarlyBinding` zeigt, dass der Aufruf überschriebener Elementfunktionen gemäß dem Typ des Objektes aufgelöst wird.

Das Auflösen von Aufrufen von Elementfunktionen basierend auf dem Typ des Objektes wird Bindung zur Compilezeit oder auch frühe Bindung genannt.

22.2 Einstieg in Polymorphie

Überschreiben von Funktionen basierend auf der Klasse von Objekten ist schon sehr schön, aber was ist, wenn die Klasse des Objektes, das eine Methode aufruft, zur Compilezeit nicht eindeutig bestimmt werden kann? Um zu demonstrieren, wie das passieren kann, lassen Sie uns das vorangegangene Programm auf eine scheinbar triviale Weise ändern. Das Ergebnis ist das Programm AmbiguousBinding, das Sie in Listing 22-2 finden.

Listing 22-2: Programm AmbiguousBinding

```
// AmbiguousBinding - die Situation wird verwirrend
//                    wenn der Typ zur Compilezeit
//                    nicht gleich dem Typ
//                    zur Laufzeit ist
#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
  public:
    double calcTuition()
    {
        return 1;
    }
};

double fn(Student& fs)
{
    // auf welche Funktion calcTuition() bezieht
    // sich der Aufruf? Welcher Wert wird
    // zurückgegeben?
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
```

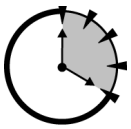
```
{
    // der folgende Ausdruck ruft
    // Student::calcTuition();
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »Aufruf durch fn() ist »
         << fn(s)
         << »\n«;

    // das ruft GraduateStudent::calcTuition();
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »Aufruf durch fn() ist »
         << fn(gs)
         << »\n«;
    return 0;
}
```

Der einzige Unterschied zwischen Listing 22-1 und 22-2 ist, dass die Aufrufe von `calcTuition( )` über eine Zwischenfunktion `fn( )` ausgeführt werden. Die Funktion `fn(Student& fs)` ist so deklariert, dass sie ein `Student`-Objekt übergeben bekommt, aber abhängig davon, wie `fn( )` aufgerufen wird, kann `fs` ein `Student` oder ein `GraduateStudent` sein. (Erinnern Sie sich? `GraduateStudent` IS_A `Student`.) Aber diese beiden Typen von Objekten berechnen ihr Schulgeld verschieden.

Weder `main( )` noch `fn( )` kümmern sich eigentlich darum, wie das Schulgeld berechnet wird. Wir hätten gerne, dass `fs.calcTuition( )` die Funktion `Student::calcTuition( )` aufruft, wenn `fs` ein `Student` ist, aber `GraduateStudent::calcTuition( )`, wenn `fs` ein `GraduateStudent` ist. Aber diese Entscheidung kann erst zur Laufzeit getroffen werden, wenn der tatsächliche Typ des übergebenen Objektes bestimmt werden kann.

Im Falle des Programms `AmbiguousBindung` sagen wir, dass der Compiletyp von `fs`, der immer `Student` ist, verschieden ist vom Laufzeittyp, der `GraduateStudent` oder `Student` sein kann.

Die Fähigkeit zu entscheiden, welche der mehrfach überladenen Elementfunktionen aufgerufen werden soll, basierend auf dem Laufzeittyp, wird als Polymorphie oder späte Bindung bezeichnet. Polymorphie kommt vom poly (=viel) und morph (=Form).



20 Min.

22.3 Polymorphie und objektorientierte Programmierung

Polymorphie ist der Schlüssel zur objektorientierten Programmierung. Sie ist so wichtig, dass Sprachen, die keine Polymorphie unterstützen, sich nicht objektorientiert nennen dürfen. Sprachen, die Klassen, aber keine Polymorphie unterstützen, werden als *objektbasierte Sprachen* bezeichnet. Ada ist ein Beispiel einer solchen Sprache.

Ohne Polymorphie hat Vererbung keine Bedeutung.

Erinnern Sie sich, wie ich Nachos im Ofen hergestellt habe? In diesem Sinne habe ich als später Binder gearbeitet. Im Rezept steht: »Nachos im Ofen erwärmen.« Da steht nicht: »Wenn der Ofen eine Mikrowelle ist, tun Sie das; wenn es ein herkömmlicher Ofen ist, tun sie das; wenn der Ofen ein Elektroofen ist, tun sie noch etwas anderes.« Das Rezept (der Code) verlässt sich auf mich (den späten Binder) zu entscheiden, welche Tätigkeit (Elementfunktion) `erwärmen` bedeutet, angewendet auf einen Ofen (die spezielle Instanz von `Oven`) oder eine ihrer Varianten (Unterklassen), wie z.B. Mikrowellen (`Microwave`). Das ist die Art und Weise, in der Leute denken, und eine Sprache in dieser Weise zu entwickeln, ermöglicht es der Sprache, besser zu beschreiben, was Leute denken.

Es gibt da noch die beiden Aspekte der Pflege und Wiederverwendbarkeit. Nehmen Sie an, ich habe dieses großartige Programm beschrieben, das die Klasse `Student` verwendet. Nach einigen Monaten des Entwurfs, der Implementierung und des Testens erstelle ich ein Release der Anwendung.

Es vergeht einige Zeit und mein Chef bittet mich, dem Programm GraduateStudent-Objekte hinzuzufügen, die sehr ähnlich zu Studenten sind, aber nicht identisch damit. Tief im Programm ruft die Funktion `someFunktion( )` die Elementfunktion `calcTuition( )` wie folgt auf:

```
void someFunction(Student &s)
{
    //... was immer sie tut ...
    s.calcTuition();
    //... wird hier fortgesetzt ...
}
```

Wenn C++ keine späte Bindung ausführen würde, müsste ich die Funktion `someFunction( )` editieren, um auch `GraduateStudent`-Objekte verarbeiten zu können. Das könnte etwa so aussehen:

```
#define STUDENT 1
#define GRADUATESTUDENT 2
void someFunction(Student &s)
{
    //... was immer sie tut ...
    // füge ein Typelement hinzu, das den
    // tatsächlichen Typ des Objekts angibt
    switch (s.type)
    {
        STUDENT:
            s.Student::calcTuition();
            break;
        GRADUATESTUDENT:
            s.GraduateStudent::calcTuition();
            break;
    }
    //... alles Weitere hier ...
}
```

Durch Verwendung des vollständigen Namens der Funktion zwingt der Ausdruck `s.GraduateStudent::calcTuition( )` ***den Aufruf, die*** `GraduateStudent`***-Version der Funktion zu verwenden, selbst wenn*** `s` ***als*** `Student` ***deklariert ist.***

Ich würde dann ein Element `type` in der Klasse einführen, das ich im Konstruktor von `Student` auf `STUDENT` setzen würde, und auf `GRADUATESTUDENT` im Konstruktor von `GraduateStudent`. Der Wert von `type` würde den Laufzeittyp von `s` darstellen. Ich würde dann den Test im Codeschnipsel einfügen, um die dem Wert dieses Elements entsprechende Funktion aufzurufen.

Das hört sich nicht schlecht an, mit Ausnahme von drei Dingen. Erstens ist das hier nur eine Funktion. Nehmen Sie an, dass `calcTuition( )` von vielen Stellen aus aufgerufen wird, und nehmen Sie an, dass `calcTuition( )` nicht der einzige Unterschied der beiden Klassen ist. Die Chancen stehen nicht sehr gut, dass ich alle Stellen finde, an denen ich etwas ändern muss.

Zweitens muss ich Code, der bereits fertiggestellt wurde, editieren (d.h. »brechen«), wodurch die Möglichkeit für neue Fehler gegeben ist. Das Editieren kann zeitaufwendig und langweilig sein, was wiederum die Gefahr von Fehlern erhöht. Irgendeine meiner Änderungen kann falsch sein oder nicht in den existierenden Code passen. Wer weiß das schon?

Schließlich, nachdem das Editieren, das erneute Debuggen und Testen abgeschlossen sind, muss ich zwei Versionen unterstützen (wenn ich nicht die Unterstützung für die Originalversion aufgeben kann). Das bedeutet zwei Quellen, die editiert werden müssen, wenn Bugs gefunden werden, und eine Art Buchhaltung, um die beiden Systeme gleich zu halten.

Was passiert, wenn mein Chef eine weitere Klasse eingefügt haben möchte? (Mein Chef ist so.) Ich muss nicht nur diesen Prozess wiederholen, ich habe dann auch drei Versionen.

Mit Polymorphie habe ich eine gute Chance, dass ich nur die neue Klasse einfügen und neu kompilieren muss. Es kann sein, dass ich die Basisklasse selber ändern muss, das ist aber wenigstens alles an einer Stelle. Änderungen an der Anwendung sollten wenige bis keine sein.

Das ist noch ein weiterer Grund, Datenelemente protected zu halten, und auf sie über als public deklarierte Elementfunktionen zuzugreifen. Datenelemente können nicht durch Polymorphie in einer Unterklasse überschrieben werden, so wie es für Elementfunktionen möglich ist.

22.4 Wie funktioniert Polymorphie?

Nach allem, was ich bisher gesagt habe, kann es verwundern, dass in C++ die frühe Bindung die Defaultmethode ist. Die Ausgabe des Programms AmbiguousBinding sieht wie folgt aus:

```
Der Wert von s.calcTuition bei
Aufruf durch fn() ist 0

Der Wert von gs.calcTuition bei
Aufruf durch fn() ist 0
```

Der Grund ist einfach. Polymorphie bedeutet ein wenig Mehraufwand, sowohl beim Speicherbedarf und beim Code, der den Aufruf ausführt. Die Erfinder von C++ waren in Sorge darüber, dass ein solcher Mehraufwand ein Grund sein könnte, die Programmiersprache C++ nicht zu verwenden, und so machten sie die frühe Bindung zur Defaultmethode.

Um Polymorphie anzuzeigen, muss der Programmierer das Schlüsselwort `virtual` verwenden, wie im Programm LateBinding zu sehen ist, das Sie in Listing 22-3 finden.

Listing 22-3: Programm LateBinding

```
// LateBinding - bei später Bindung wird die
//               Entscheidung, welche der
//               überschriebenen Funktionen
//               aufgerufen wird, zur Laufzeit
//               getroffen
#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    virtual double calcTuition()
    {
        return 0;
    }
};
class GraduateStudent : public Student
{
  public:
    virtual double calcTuition()
    {
        return 1;
    }
};

double fn(Student& fs)
{
    // weil calcTuition() virtual deklariert ist,
    // wird der Laufzeittyp von fs verwendet, um
    // den Aufruf aufzulösen
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // fn() mit einem Student-Objekt
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(s)
         << »\n\n«;

    // der folgende Ausdruck ruft
    // fn() mit einem GraduateStudent-Objekt
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(gs)
         << »\n\n«;
    return 0;
}
```

Das Schlüsselwort `virtual`, das der Deklaration von `calcTuition( )` zugefügt wurde, erzeugt eine virtuelle Elementfunktion. Das bedeutet, dass Aufrufe von `calcTuition( )` spät gebunden werden, wenn der Typ des aufrufenden Objektes nicht zur Compilezeit bestimmt werden kann.

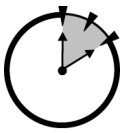
Das Programm `LateBindung` enthält den gleichen Aufruf der Funktion `fn( )` wie in den beiden früheren Versionen. In dieser Version geht der Aufruf von `calcTuition( )` an `Student::calcTuition( )`, wenn `fs` ein `Student` ist und an `GraduateStudent::calcTuition( )`, wenn `fs` ein `GraduateStudent` ist.

Die Ausgabe von `LateBinding` sehen Sie unten. Die Funktion `calcTuition( )` als virtuell zu deklarieren, lässt `fn( )` Aufrufe anhand des Laufzeittyps auflösen.

```
Der Wert von s.calcTuition bei
virtuellem Aufruf durch fn() ist 0

Der Wert von gs.calcTuition bei
virtuellem Aufruf durch fn() ist 1
```

Bei der Definition einer virtuellen Elementfunktion steht das Schlüsselwort `virtual` nur bei der Deklaration und nicht bei der Definition, wie im folgenden Beispiel zu sehen ist:

```
class Student
{
  public:
    // deklariere als virtual
    virtual double calcTuition()
    {
        return 0;
    }
};

// 'virtual' kommt in der Definition nicht vor
double Student::calcTuition()
{
    return 0;
}
```



10 Min.

22.5 Was ist eine virtuelle Funktion nicht?

Nur weil Sie denken, dass ein bestimmter Funktionsaufruf spät gebunden wird, bedeutet das nicht, dass dies auch der Fall ist. C++ erzeugt beim Kompilieren keine Hinweise darauf, welche Aufrufe es früh und welche es spät bindet.

Die kritischste Sache ist die, dass alle Elementfunktionen, die in Frage kommen, identisch deklariert sind, ihren Rückgabetype eingeschlossen. Wenn sie nicht identisch deklariert sind, werden die Elementfunktionen nicht mittels Polymorhpie überschrieben, ob sie nun als virtual deklariert sind oder nicht. Betrachen sie den folgenden Codeschnipsel:

```
#include <iostream.h>
class Base
{
  public:
    virtual void fn(int x)
    {
        cout << »In Base int x = »
             << x << »\n«;
    }
};
class SubClass : public Base
{
  public:
    virtual void fn(float x)
    {
        cout << »In SubClass, float x = »
             << x << »\n«;
    }
};

void test(Base &b)
{
    int i = 1;
    b.fn(i);          // nicht spät gebunden
    float f = 2.0;
    b.fn(f);          // und der Aufruf auch nicht
}
```

`fn( )` in `Base` ist als `fn(int)` deklariert, während die Version in der Unterklasse als `fn(float)` deklariert ist. Weil die Funktionen verschiedene Argumente haben, gibt es keine Polymorphie. Der erste Aufruf geht an `Base::fn(int)` – das ist nicht verwunderlich, weil `b` vom Typ `Base` und `i` ein `int` ist. Doch auch der nächste Aufruf geht an `Base::fn(int)`, nachdem `float` in `int` konvertiert wurde. Es wird kein Fehler erzeugt, weil dieses Programm legal ist (abgesehen von der Warnung, die Konvertierung von `f` betreffend). Die Ausgabe eines Aufrufs von `test( )` zeigt keine Polymorphie:

```
In Base, int x = 1
In Base, int x = 2
```

Die Argumente passen nicht exakt, es gibt keine späte Bindung – mit einer Ausnahme: Wenn die Elementfunktion in der Basisklasse einen Zeiger oder eine Referenz auf ein Objekt der Basisklasse zurückgibt, kann eine überschriebene Elementfunktion in einer Unterklasse einen Zeiger oder eine Referenz auf ein Objekt der Unterklasse zurückgeben. Mit anderen Worten, das Folgende ist erlaubt:

```
class Base
{
  public:
    Base* fn();
};

class Subclass : public Base
{
  public:
    Subclass* fn();
};
```

In der Praxis ist das ganz natürlich. Wenn eine Funktion mit `Subclass`-Objekten umgeht, scheint es natürlich zu sein, dass sie damit fortfährt.

22.6 Überlegungen zu virtual

Den Namen der Klasse im Aufruf anzugeben, erzwingt frühe Bindung. Der folgende Aufruf geht z.B. an `Base::fn( )`, weil der Programmierer das so ausgedrückt hat, auch wenn `fn( )` als virtual deklariert ist.

```
void test(Base &b)
{
    b.Base::fn(); // wird nicht spät gebunden
}
```

Eine als virtual deklarierte Funktion kann nicht inline sein. Um eine Inline-Funktion zu expandieren, muss der Compiler zur Compilezeit wissen, welche Funktion expandiert werden soll. Daher waren auch alle Elementfunktionen, die Sie in den bisherigen Beispielen gesehen haben, outline deklariert.

Konstruktoren können nicht virtual sein, weil es kein (fertiges) Objekt gibt, das zur Typbestimmung verwendet werden kann. Zum Zeitpunkt, an dem der Konstruktor aufgerufen wird, ist der Speicher, der von dem Objekt belegt wird, nur eine formlose Masse. Erst nachdem der Konstruktor fertig ist, ist das Objekt ein Element der Klasse im eigentlichen Sinne.

Im Vergleich dazu sollten Destruktoren normalerweise als virtual deklariert werden. Wenn nicht, gehen Sie das Risiko ein, dass ein Objekt nicht korrekt vernichtet wird, wie in folgender Situation:

```
class Base
{
  public:
    ~Base();
};
class SubClass : public Base
{
  public:
    ~SubClass();
};
void finishWithObject(Base *pHeapObject)
{
    // ... arbeite mit Objekt ...
    // jetzt gib es an den Heap zurück
    delete pHeapObject;  // ruft ~Base() auf,
}                        // unabhängig von Laufzeit-
                         // typ von pHeapObject
```

Wenn der Zeiger, der an `finishWithObject( )` tatsächlich auf ein Objekt aus der Klasse `SubClass` zeigt, wird der `SubClass`-Destruktor trotzdem nicht korrekt aufgerufen. Den Destruktor virtuell zu deklarieren, löst das Problem.

Wann würden Sie also einen Destruktor nicht virtuell deklarieren? Es gibt nur eine solche Situation. Ich habe bereits erwähnt, dass virtuelle Funktionen einen gewissen »Mehraufwand« bedeuten. Lassen Sie mich ein wenig genauer sein. Wenn der Programmierer die erste virtuelle Funktion in einer Klasse definiert, fügt C++ einen zusätzlichen, versteckten Zeiger hinzu – nicht einen Zeiger pro virtueller Funktion, nur einen Zeiger, falls die Klasse mindestens eine virtuelle Funktion besitzt. Eine Klasse, die keine virtuellen Funktionen besitzt (und keine virtuellen Funktionen von einer Basisklasse erbt), besitzt keinen solchen Zeiger.

Nun, ein Zeiger klingt nicht nach sehr viel und ist es auch nicht, es sei denn die folgenden zwei Bedingungen sind erfüllt:

- Die Klasse hat nicht viele Datenelemente (so dass ein Zeiger viel ist im Vergleich zum Rest).
- Sie beabsichtigen, viele Objekte dieser Klasse zu erzeugen (ansonsten macht der zusätzliche Speicher keinen Unterschied).

0 Min.

Wenn diese beiden Bedingungen erfüllt sind und Ihre Klasse nicht bereits eine virtuelle Elementfunktion besitzt, können Sie Ihren Destruktor als nicht virtual deklarieren.

Normalerweise sollten Sie aber den Destruktor virtual deklarieren. Wenn Sie das mal nicht tun, dokumentieren Sie die Gründe dafür!

Zusammenfassung

Vererbung an sich ist schön, ist aber begrenzt in ihren Möglichkeiten. In Kombination mit Polymorphie, ist Vererbung ein mächtiges Programmierwerkzeug.

- Elementfunktionen in einer Klasse können Elementfunktionen überschreiben, die in der Basisklasse definiert sind. Aufrufe dieser Funktionen werden zur Compilezeit aufgelöst basierend auf der zur Compilezeit bekannten Klasse. Das wird frühe Bindung genannt.
- Eine Elementfunktion kann als virtual deklariert werden, wodurch Aufrufe basierend auf dem Laufzeittyp aufgelöst werden. Das wird Polymorphie oder späte Bindung genannt.
- Aufrufe, von denen bekannt ist, dass der Laufzeittyp und der Compiletyp gleich sind, werden früh gebunden, unabhängig davon, ob die Elementfunktion als virtual deklariert ist oder nicht.

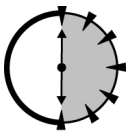
Selbsttest

1 Was ist Polymorphie? (Siehe »Einstieg in Polymorphie«)
2. Was ist ein anderes Wort für Polymorphie? (Siehe »Einstieg in Polymorphie«)
3. Was ist die Alternative und wie wird sie genannt? (Siehe »Elementfunktionen überschreiben«)
4. Nennen Sie drei Gründe, weshalb C++ Polymorphie enthält. (Siehe »Polymorphie und objektorientierte Programmierung«)
5. Welches Schlüsselwort wird verwendet, um Elementfunktion polymorph zu deklarieren? (Siehe »Wie funktioniert Polymorphie?«)

Abstrakte Klassen und Faktorieren

Checkliste

- ☑ Gemeinsame Eigenschaften in eine Basisklasse faktorieren
- ☑ Abstrakte Klassen zur Speicherung faktorierter Informationen nutzen
- ☑ Abstrakte Klassen und dynamische Typen



30 Min.

Bis jetzt haben wir gesehen, wie Vererbung benutzt werden kann, um existierende Klassen für neue Anwendungen zu erweitern. Vererbung verlangt vom Programmierer die Fähigkeit, gleiche Eigenschaften verschiedener Klassen zu kombinieren; dieser Prozess wird *Faktorieren* genannt.

23.1 Faktorieren

Um zu sehen, wie Faktorieren funktioniert, lassen Sie uns die beiden Klassen `Checking` (Girokonto) und `Savings` (Sparkonto) in einem hypothetischen Banksystem betrachten. Diese sind in Abbildung 23.1 grafisch dargestellt.

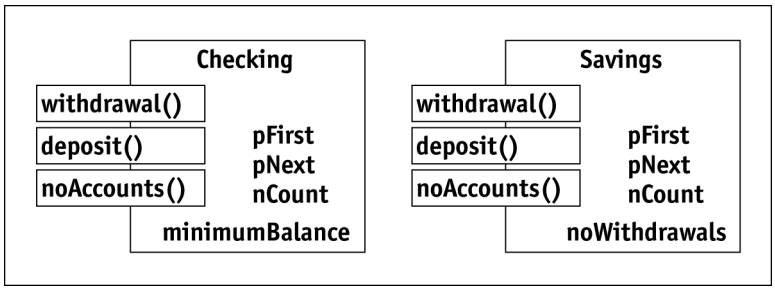


Abbildung 23.1: Unabhängige Klassen Checking und Savings.

Um diese Abbildung und die folgenden Abbildungen lesen zu können, halten Sie im Gedächtnis, dass

- die große Box eine Klasse ist, mit dem Klassennamen ganz oben,
- die Namen in den Boxen Elementfunktionen sind,
- die Namen ohne Box Datenelemente sind,
- die Namen, die teilweise außerhalb der Boxen liegen, öffentlich zugängliche Elemente sind; die anderen protected deklariert sind.
- ein dicker Pfeil die Beziehung IS_A repräsentiert und
- ein dünner Pfeil die Beziehung HAS_A repräsentiert.

Abbildung 23.1 zeigt, dass die Klassen `Checking` und `Savings` vieles gemein haben. Weil sie jedoch nicht identisch sind, müssen es zwei getrennte Klassen bleiben. Dennoch sollte es einen Weg geben, Wiederholungen zu vermeiden.

Wir könnten eine der Klassen von der anderen erben lassen. `Savings` hat ein Extraelement, es macht daher mehr Sinn, `Savings` von `Checking` abzuleiten, wie Sie in Abbildung 23.2 sehen, als umgekehrt. Die Klasse wird vervollständigt durch das Hinzufügen des Datenelementes `noWithdrawals` und dem virtuellen Überladen der Elementfunktion `withdrawal( )`.

Obwohl die Lösung Arbeit einspart, ist sie nicht zufriedenstellend. Das Hauptproblem ist, dass es die Wahrheit falsch darstellt. Diese Vererbungsbeziehung impliziert, dass ein Sparkonto (`Savings`) ein spezieller Typ eines Girokontos (`Checking`) ist, was nicht der Fall ist.

»Na und?« werden Sie sagen. »Es funktioniert und spart Aufwand.« Das ist wahr, aber meine Vorbehalte sind mehr als sprachliche Trivialitäten. Solche Fehldarstellungen verwirren den Programmierer, den heutigen und den von morgen. Eines Tages wird ein Programmierer, der sich mit dem Programm nicht auskennt, das Programm lesen, und verstehen müssen, was der Code macht. Irreführende Tricks sind schwer zu durchschauen und zu verstehen.

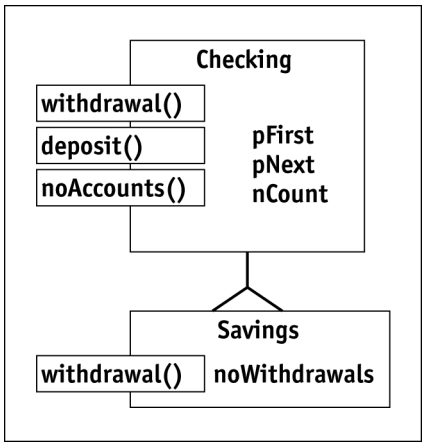


Abbildung 23.2: Savings implementiert als Unterklasse von Checking

Außerdem können solche Fehldarstellungen zu späteren Problemen führen. Nehmen Sie z.B. an, dass die Bank ihre Policen für Girokonten ändert. Sagen wir, die Bank entscheidet, dass sie eine Bearbeitungsgebühr nur dann verlangt, wenn der mittlere Kontostand im Monat unter einem gegebenen Wert liegt.

Eine solche Änderung kann mit minimalen Änderungen an der Klasse `Checking` leicht durchgeführt werden. Wir müssen ein neues Datenelement in die Klasse `Checking` einführen, das wir `minimumBalance` nennen wollen.

Das erzeugt aber ein Problem. Weil Savings von Checking erbt, bekommt Savings ebenfalls ein solches Datenelement. Die Klasse hat aber für ein solches Element keine Verwendung, weil der minimale Kontostand das Sparkonto nicht beeinflusst. Ein zusätzliches Datenelement macht nicht so viel aus, aber es verwirrt.

Änderungen wie diese akkumulieren sich. Heute ist es ein zusätzliches Datenelement, morgen ist es eine geänderte Elementfunktion. Schließlich hat die Klasse `Savings` einen großen Ballast, der nur auf die Klasse `Checking` angewendet werden kann.

Wie vermeiden wir dieses Problem? Wir können beide Klassen auf einer neuen Klasse basieren lassen, die speziell für diesen Einsatz gebaut ist; lassen Sie uns diese Klasse `Account` (=Konto) nennen. Diese Klasse enthält alle Eigenschaften, die `Savings` und `Checking` enthalten, wie in Abbildung 1.3 zu sehen ist.

Wie löst das unser Problem? Erstens ist das eine viel treffendere Beschreibung der realen Welt (was immer das ist). In meinem Konzept gibt es etwas, das als Konto bezeichnet wird. Girokonto und Sparkonto sind Spezialisierungen dieses fundamentaleren Konzeptes.

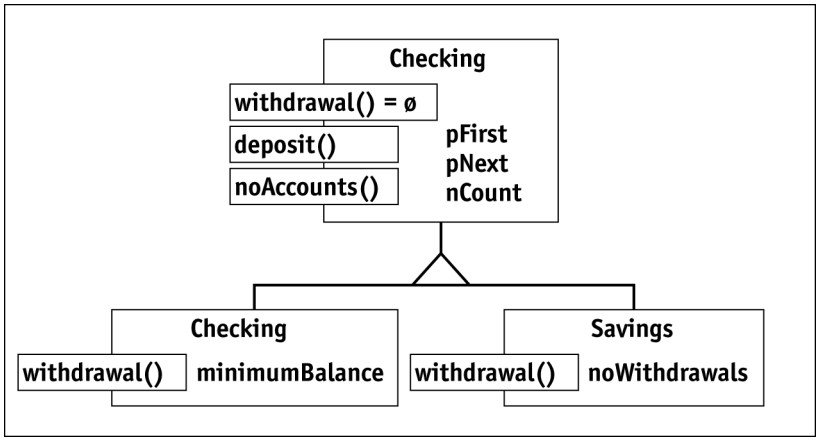


Abbildung 23.3: Checking und Savings auf Klasse Account basieren lassen

Zusätzlich bleibt die Klasse `Savings` von Änderungen an der Klasse `Checking` unberührt (und umgekehrt). Wenn die Bank eine grundlegende Änderung an allen Konten durchführen möchte, können wir die Klasse `Account` modifizieren und alle abgeleiteten Klassen erben diese Änderung automatisch. Aber wenn die Bank ihre Policen nur für Girokonten ändert, bleibt die Klasse `Savings` von dieser Änderung verschont.

Dieser Prozess, gleiche Eigenschaften aus ähnlichen Klassen zu extrahieren, wird als `Faktorieren` bezeichnet. Das ist ein wichtiges Feature objektorientierter Sprachen aus den bereits genannten Gründen, plus einem neuen: Reduktion von Redundanz.

In Software ist nutzlose Masse eine üble Sache. Je mehr Code Sie generieren, desto mehr müssen Sie auch debuggen. Es lohnt nicht, Nachtschichten einzulegen, um cleveren Code zu generieren, der hier und da ein paar Zeilen Code einspart – diese Art Schlauheit entpuppt sich oft als Bumerang. Aber das Faktorieren redundanter Information durch Vererbung kann den Programmieraufwand tatsächlich reduzieren.

Faktorieren ist nur zulässig, wenn die Vererbungsbeziehung der Realität entspricht. Zwei Klassen `Mouse` ***und*** `Joystick` ***zu faktorieren ist legitim, weil es beide Klassen sind, die Zeigerhardware beschreiben. Zwei Klassen Mouse und Display zu faktorieren, weil sie elementare Systemfunktionen des Betriebssystems benutzen, ist nicht legitim –*** `Maus` ***und*** `Bildschirm` ***teilen keine Eigenschaft in der realen Welt.***

Faktorieren kann, und wird es auch in der Regel, zu mehreren Abstraktionsstufen führen. Ein Programm, das z.B. für eine fortschrittlichere Bank geschrieben wurde, könnte eine Klassenstruktur enthalten, wie in Abbildung 23.4 zu sehen ist.

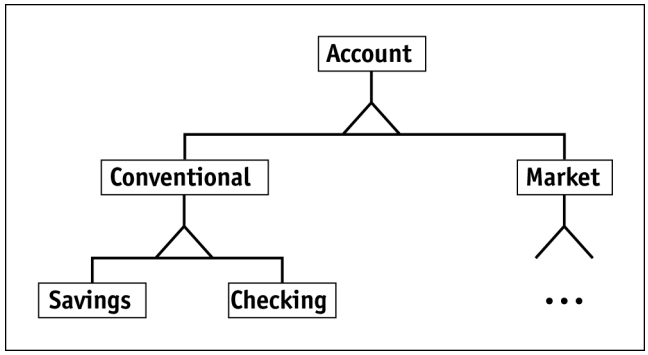


Abbildung 23.4: Eine weiter entwickelte Hierarchie für eine Bank.

Es wurde eine weitere Klasse zwischen den Klassen `Checking` und `Savings` und der allgemeineren Klasse `Account` eingeführt. Diese Klasse `Conventional` enthält die Features konventioneller Kontos. Andere Kontotypen wie z.B. Aktiondepots, sind ebenso vorgesehen.

Solche mehrarmigen Klassenstrukturen sind üblich und anzustreben, so lange ihre Beziehungen die Wirklichkeit widerspiegeln. Es gibt jedoch nicht nur eine korrekte Klassenhierarchie für eine gegebene Menge von Klassen.

Nehmen Sie an, dass unsere Bank es ihren Kunden ermöglicht, Girokonten und Aktiendepots online zu verwalten. Transaktionen für andere Kontotypen können nur bei der Bank getätigt werden. Obwohl die Klassenstruktur in Abbildung 23.4 natürlich erscheint, ist die Hierarchie in Abbildung 23.5 mit dieser Information ebenfalls gerechtfertigt. Der Programmierer muss entscheiden, welche Klassenhierarchie am besten zu den Daten passt, und zu der saubersten und natürlichsten Implementierung führen wird.

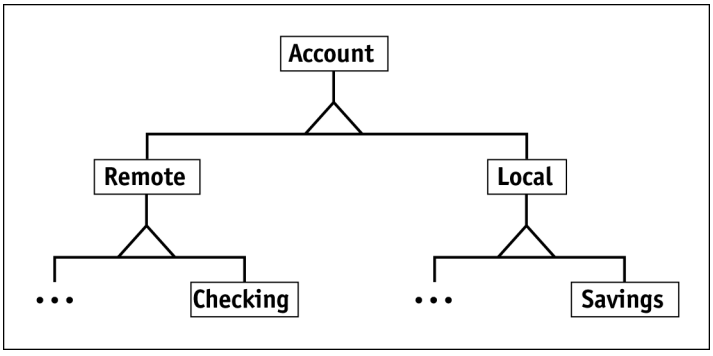
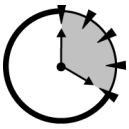


Abbildung 23.5: Eine alternative Klassenhierarchie zu Abbildung 23.4



20 Min.

23.2 Abstrakte Klassen

So sehr Faktorieren auch den Intellekt befriedigt, bringt es ein Problem mit sich. Lassen Sie uns ein weiteres Mal auf das Kontobeispiel zurückkommen, insbesondere auf die gemeinsame Basisklasse `Account`. Lassen Sie uns eine Minute überlegen, wie wir die verschiedenen Elementfunktionen von `Account` definieren würden.

Die meisten Elementfunktionen von `Account` sind kein Problem, weil beide Kontotypen sie auf die gleiche Weise implementieren. Bei `withdrawal( )` ist das anders. Die Regeln zum Abheben sind bei Girokonten und Sparkonten verschieden. Somit würden wir erwarten, dass `Savings::withdrawal( )` und `Checking::withdrawal( )` unterschiedlich implementiert sind. Aber die Frage ist ja, wie implementieren wir dann `Account::withdrawal( )`?

»Kein Problem«, werden Sie sagen. »Gehen Sie einfach zu Ihrer Bank und fragen Sie dort »Wie sind die Regeln für das Abheben von Konten?« Die Antwort ist »Welche Art Konto?« Ein ratloser Blick.

Das Problem ist, dass die Frage keinen Sinn macht. Es gibt keine Sache »einfach Konto«. Alle Konten (in diesem Beispiel) sind entweder Girokonten oder Sparkonten. Das Konzept Konto ist ein abstraktes, das gleiche Eigenschaften der konkreten Klassen faktoriert. Es ist jedoch unvollständig, weil es die kritische Eigenschaft `withdrawal( )` nicht besitzt. (Wenn wir zu den Details kommen, werden wir weitere Eigenschaften finden, die einem einfachen Konto fehlen.)

Lassen Sie mich ein Beispiel aus der Tierwelt entleihen. Wir können die verschiedenen Spezies der warmblütigen lebendgebärenden Tiere unterscheiden und daraus schließen, dass es ein Konzept Säugetiere gibt. Wir können von dieser Klasse *Säugetier* Klassen ableiten wie *Hund*, *Katze* und *Mensch*. Es ist jedoch nicht möglich, irgendwo etwas zu finden, das ein reines Säugetier ist, d.h. ein Säugetier, das nicht zu einer der Spezies gehört. Säugetier ist ein Konzept auf hohem Abstraktionsniveau – es gibt keine Instanz von Säugetier.

Das Konzept Säugetier unterscheidet sich grundlegend vom Konzept Hund. »Hund« ist ein Name, den wir einem existierenden Objekt gegeben haben. Es gibt nichts in der realen Welt was nur Säugetier ist.

Wir möchten nicht, dass der Programmierer ein Objekt `Account` (=Konto) oder eine Klasse *Mammal* (=Säugetier) erzeugt, weil wir nicht wissen, was wir damit anfangen sollen. Um diesem Problem zu begegnen, erlaubt es C++ dem Programmierer, eine Klasse zu deklarieren, von der kein Objekt instanziiert werden kann. Der einzige Sinn einer solchen Klasse ist, dass sie vererben kann.

Eine Klasse, die nicht instanziiert werden kann, heißt *abstrakte* Klasse.

23.2.1 Deklaration einer abstrakten Klasse

Eine abstrakte Klasse ist eine Klasse mit einer oder mehreren rein virtuellen Funktionen. Eine *rein virtuelle Funktion* ist eine virtuelle Elementfunktion, die so markiert ist, dass sie keine Implementierung besitzt.

Eine rein virtuelle Funktion hat keine Implementierung, weil wir nicht wissen, wie wir sie implementieren sollen. Z.B. wissen wir nicht, wie wir `withdrawal( )` in einer Klasse `Account` ausführen sollen. Das macht einfach keinen Sinn. Wir können jedoch nicht einfach die Definition von `withdra-`

wal( ) weglassen, weil C++ annehmen wird, dass wir vergessen haben, die Funktion zu definieren und uns einen Linkfehler ausgeben wird, der uns mitteilt, dass eine Funktion fehlt (wahrscheinlich vergessen).

Die Syntax zur Deklaration einer rein virtuellen Funktion – die C++ mitteilt, dass die Funktion keine Definition hat – wird in folgender Klasse Account demonstriert:

```
// Account - das ist eine abstrakte Basisklasse
//          für alle Kontoklassen
class Account
{
  public:
    Account(unsigned nAccNo);

    // Zugriffsfunktionen
    int accountNo();
    Account* first();
    Account* next();

    // Transaktionsfunktionen
    virtual void deposit(float fAmount) = 0;
    virtual void withdrawal(float fAmount) = 0;

  protected:
    // speichere Kontoobjekte in einer Liste, damit
    // es keine Beschränkung der Anzahl gibt
    static Account* pFirst;
           Account* pNext;

    // alle Konten haben eine eindeutige Nummer
    unsigned nAccountNumber;
};
```

Die =0 hinter der Deklaration von deposit( ) und withdrawal( ) zeigt an, dass der Programmierer nicht beabsichtigt, diese Funktionen zu definieren. Die Deklaration ist ein Platzhalter für die Unterklassen. Von den konkreten Unterklassen von Account wird erwartet, dass sie diese Funktionen mit konkreten Funktionen überladen.

Eine konkrete Elementfunktion ist eine Funktion, die nicht rein virtuell ist. Alle Elementfunktion vor dieser Sitzung waren konkret.

Obwohl diese Notation, die =0 verwendet, anzeigt, dass eine Elementfunktion abstrakt ist, bizarr ist, bleibt sie doch so. Es gibt einen obskuren Grund dafür, wenn auch nicht gerade eine Rechtfertigung, aber das geht über den Bereich dieses Buches hinaus.

Eine abstrakte Klasse kann nicht mit einem Objekt instanziiert werden. D.h. sie können kein Objekt aus einer abstrakten Klasse anlegen. Z.B. ist das Folgende nicht möglich:

```
void fn()
{
    // deklariere ein Konto
    Account acnt(1234);  // das ist nicht erlaubt
    acnt.withdrawal(50); // was soll das tun?
}
```

Wenn die Deklaration erlaubt wäre, würde das resultierende Objekt unvollständig sein, und einige Eigenschaften vermissen lassen. Was soll z.B. der obige Aufruf von `acnt.withdrawal(50)` tun? Es gibt keine Funktion `Account::withdrawal( )`.

Abstrakte Klassen dienen als Basisklassen für andere Klassen. Ein `Account` enthält alle die Eigenschaften, die wir einem generischen Konto zuschreiben, die Möglichkeit des Abhebens und des Einzahlens eingeschlossen. Wir können nur nicht definieren, wie ein generisches Konto solche Dinge ausführt – es bleibt den Unterklassen, das zu definieren. Anders ausgedrückt, ein Konto ist so lange kein Konto, bis der Benutzer Einzahlungen und Abhebungen machen kann, selbst wenn solche Operationen nur in speziellen Kontotypen definiert werden können, wie z.B. Girokonten und Sparkonten.

Wir können weitere Typen vom Konten durch Ableitung von `Account` erzeugen, aber sie können nicht durch ein Objekt instanziiert werden, so lange sie abstrakt bleiben.

23.2.2 Erzeugung einer konkreten Klasse aus einer abstrakten Klasse

Die Unterklasse einer abstrakten Klasse bleibt abstrakt, bis alle virtuellen Funktionen überladen sind. Die folgende Klasse `Savings` ist nicht abstrakt, weil sie die rein virtuellen Funktionen `deposit( )` und `withdrawal( )` mit perfekten Definitionen überlädt.

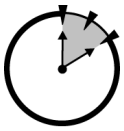
```
// Account - das ist eine abstrakte Basisklasse
//          für alle Kontoklassen
class Account
{
  public:
    Account(unsigned nAccNo);

    // Zugriffsfunktionen
    int accountNo();
    Account* first();
    Account* next();

    // Transaktionsfunktionen
    virtual void deposit(float fAmount) = 0;
    virtual void withdrawal(float fAmount) = 0;

  protected:
    // speichere Kontoobjekte in einer Liste, damit
    // es keine Beschränkung der Anzahl gibt
    static Account* pFirst;
           Account* pNext;

    // alle Konten haben eine eindeutige Nummer
    unsigned nAccountNumber;
};
```

```
// Savings - implementiert das Konzept Account
class Savings : public Account
{
  public:
    // Konstruktor - Sparbücher werden mit einem
    //             initialen Kontostand erzeugt
    Savings(unsigned nAccNo,
        float fInitialBalance)
        : Account(nAccNo)
    {
        fBalance = fInitialBalance;
    }

    // Sparkonten wissen, wie diese Operationen
    // ausgeführt werden
    virtual void deposit(float fAmount);
    virtual void withdrawal(float fAmount);

  protected:
    float fBalance;
};

// deposit and withdrawal - definiert die Standard-
//                          kontooperationen für
//                          Sparkonten
void Savings::deposit(float fAmount)
{
    // ... die Funktion ...
}
void Savings::withdrawal(float fAmount)
{
    // ... die Funktion ...
}
```

Ein Objekt der Klasse `Savings` weiß, wie Einzahlungen und Abhebungen ausgeführt werden, wenn sie aufgerufen werden. Das Folgende macht also Sinn:

```
void fn()
{
    Savings s(1234);
    s.deposit(100.0);
}
```

Die Klasse `Account` ***hat einen Konstruktor, obwohl sie abstrakt ist. Alle Konten werden mit einer ID erzeugt. Die konkrete Kontoklasse*** `Savings` ***übergibt die ID an die Basisklasse*** `Account`***, während Sie den initialen Kontostand selbst übernimmt. Das ist Teil unseres Objektmodells – die Klasse*** `Account` ***enthält das Element für die Kontonummer, somit wird es*** `Account` ***überlassen, dieses Feld zu initialisieren.***



10 Min.

23.2.3 Warum ist eine Unterklasse abstrakt?

Eine Unterklasse einer abstrakten Klasse kann abstrakt bleiben. Stellen Sie sich vor, wir hätten eine weitere Zwischenklasse in die Klassenhierarchie eingefügt. Nehmen Sie z.B. an, dass meine Bank so etwas wie ein Geldkonto eingeführt hat.

Geldkonten sind Konten, in denen Guthaben in Geld und nicht z.B. in Wertpapieren vorliegt. Girokonten und Sparkonten sind Geldkonten. Alle Einzahlungen auf Geldkonten werden bei meiner Bank gleich gehandhabt; Sparkonten berechnen jedoch nach den ersten fünf Abhebungen Gebühren, während Girokonten keine Gebühren für etwas erheben.

Mit diesen Definitionen kann die Klasse `CashAccount` die Funktion `deposit( )` implementieren, weil die Operation wohldefiniert und für alle Geldkonten gleich ist; `CashAccount` kann jedoch `withdawal( )` nicht implementieren, weil unterschiedliche Geldkonten diese Operation unterschiedlich ausführen.

In C++ sehen die Klassen `CashAccount` und `Savings` wie folgt aus:

```
// CashAccount - ein Geldkonto speichert Geldwerte
//               und nicht z.B. Wertpapiere.
//               Geldkonten erfordern Geldangaben,
//               alle Einzahlungen werden gleich
//               behandelt. Abhebungen werden von
//               verschiedenen Geldkontoformen ver-
//               schieden gehandhabt.
class CashAccount : public Account
{
  public:
    CashAccount(unsigned nAccNo,
                float fInitialBalance)
        : Account(nAccNo)
    {
        fBalance = fInitialBalance;
    }

    // Transaktionsfunktionen
    // deposit - alle Geldkonten erwarten
    //           Einzahlungen als Betrag
    virtual void deposit(float fAmount)
    {
        fBalance += fAmount;
    }

    // Zugriffsfunktionen
    float balance()
    {
        return fBalance;
    }

  protected:
    float fBalance;
};

// Savings - ein Sparkonto ist ein Geldkonto; die
//           Operation des Abhebens ist wohldefiniert
class Savings : public CashAccount
```

```
{
  public:
    Savings(unsigned nAccNo,
            float fInitialBalance = 0.0F)
        : CashAccount(nAccNo, fInitialBalance)
    {
        // ... was immer Savings tun muss,
        // was ein Account noch nicht getan hat ...
    }

    // ein Sparkonto weiß, wie Abheben funktionieren
    virtual void withdrawal(float fAmount);
};

// fn - eine Testfunktion
void fn()
{
    // eröffne ein Sparkonto mit $200 darauf
    Savings savings(1234, 200);

    // Einzahlung $100
    savings.deposit(100);

    // und $50 abheben
    savings.withdrawal(50);
}
```

Die Klasse `CashAccout` bleibt abstrakt, weil sie die Funktion `deposit( )`, aber nicht die Funktion `withdrawal( )` überlädt. `Savings` ist konkret, weil sie die verbleibende rein virtuelle Elementfunktion überlädt.

Die Testfunktion `fn( )` erzeugt ein `Savings`-Objekt, tätigt eine Einzahlung und dann eine Abhebung.

Ursprünglich musste jede rein virtuelle Funktion in einer Unterklasse überladen werden, selbst wenn die Funktion mit einer weiteren rein virtuellen Funktion überladen wurde. Schließlich haben die Leute festgestellt, dass das genauso dumm ist, wie es sich anhört, und haben diese Forderung fallen gelassen. Weder Visual C++ noch GNU C++ stellen diese Forderung, ältere Compiler tun das möglicherweise.

23.2.4 Ein abstraktes Objekt an eine Funktion übergeben

Obwohl Sie keine abstrakte Klasse instanziieren können, ist es möglich, einen Zeiger oder eine Referenz auf eine abstrakte Klasse zu deklarieren. Mit Polymorphie ist das aber gar nicht so verrückt, wie es klingt. Betrachen Sie den folgenden Codeschnipsel:

```
void fn(Account* pAccount){ // das ist legal
{
    pAccount->withdrawal(100.0);
}
```

```
void otherFn()
{
    Savings s;

     // ist legal weil Savings IS_A Account
    fn(&s);
}
```

Hier wird `pAccount` als Zeiger auf `Account` deklariert. Die Funktion `fn( )` darf `pAccount->withdrawal( )` aufrufen, weil alle Konten wissen, wie sie Auszahlungen vornehmen. Es ist aber auch klar, dass die Funktion beim Aufruf die Adresse eines Objektes einer nichtabstrakten Unterklasse übergeben bekommt, wie z.B. `Savings`.

Es ist wichtig, hier darauf hinzuweisen, dass jedes Objekt, das `fn( )` übergeben wird, entweder aus `Savings` kommt, oder aus einer anderen nichtabstrakten Unterklasse von `Account`. Die Funktion `fn( )` kann sicher sein, dass wir niemals ein Objekt aus der Klasse `Account` übergeben werden, weil wir so ein Objekt nie erzeugen können. Das Folgende kann also nie passieren, weil C++ es nicht erlauben würde:

```
void otherFn()
{
    // das Folgende ist nicht erlaubt, weil Account
    // eine abstrakte Klasse ist
    Account a;

    fn(&a);
}
```

Der Schlüssel ist, dass es `fn( )` erlaubt war, `withdrawal( )` mit einem abstrakten `Account`-Objekt aufzurufen, weil jede konkrete Unterklasse von `Account` weiß, wie sie die Operation `withdrawal( )` ausführen muss.

Eine rein virtuelle Funktion stellt ein Versprechen dar, eine bestimmte Eigenschaft in den konkreten Unterklassen zu implementieren.

23.2.5 Warum werden rein virtuelle Funktionen benötigt?

Wenn `withdrawal( )` nicht in der Basisklasse `Account` definiert werden kann, warum lässt man sie dann dort nicht weg? Warum definiert man die Funktion nicht in `Savings` und lässt sie aus `Account` heraus? In vielen objektorientierten Sprachen können Sie das nur so machen. Aber C++ möchte in der Lage sein, zu überprüfen, dass Sie wirklich wissen, was Sie tun.

C++ ist eine streng getypte Sprache. Wenn Sie eine Elementfunktion ansprechen, besteht C++ darauf, dass sie beweisen, dass die Elementfunktion in der von Ihnen angegebenen Klasse existiert. Das verhindert unglückliche Laufzeitüberraschungen, wenn eine referenzierte Elementfunktion nicht gefunden werden kann.

Lassen Sie uns die folgenden kleineren Änderungen an Account ausführen, um das Problem zu demonstrieren:

```
class Account
{
    // wie zuvor, doch ohne Deklaration von
    // withdrawal()
};
class Savings : public Account
{
  public:
    virtual void withdrawal(float fAmount);
};

void fn(Account* pAcc)
{
    // hebe etwas Geld ab
    pAcc->withdrawal(100.00F); // das ist nicht
                               // erlaubt, weil
                               // withdrawal() nicht
                               // Element der Klasse
                               // Account ist
};
int otherFn()
{
    Savings s; // eröffne ein Konto
    fn(&s);
    //... Fortsetzung ...
}
```

Die Funktion `otherFn( )` arbeitet wie zuvor. Wie vorher auch, versucht die Funktion `fn( )` die Funktion `withdrawal( )` mit dem `Account`-Objekt aufzurufen, das sie erhält. Weil die Funktion `withdrawal( )` kein Element von `Account` ist, erzeugt der Compiler jedoch einen Fehler.

In diesem Fall hat die Klasse `Account` kein Versprechen abgegeben, eine Elementfunktion `withdrawal( )` zu implementieren. Es könnte eine konkrete Unterklasse von `Account` geben, die keine solche Operation `withdrawal( )` definiert. In diesem Fall hätte der Aufruf `pAcc->withdrawal( )` keinen Zielort – das ist eine Möglichkeit, die C++ nicht akzeptieren kann.

0 Min.

Zusammenfassung

Klassen von Objekten auf der Basis wachsender Gemeinsamkeiten in Hierarchien aufzuteilen,, wird als Faktorieren bezeichnet. Faktorieren führt fast unausweichlich zu Klassen, die eher konzeptionell als konkret sind. Ein Mensch ist ein Primat, ist ein Säugetier; die Klasse Mammal (=Säugetier) ist jedoch konzeptionell und nicht konkret – es gibt keine Instanz von Mammal, die nicht zu einer bestimmten Spezies gehört.

Sie haben ein Beispiel dafür mit der Klasse `Account` gesehen. Während es Sparkonten und Girokonten gibt, gibt es kein Objekt, das einfach nur ein Konto ist. In C++ sagen wir, dass `Account` eine abstrakte Klasse ist. Eine Klasse wird abstrakt, sobald eine ihrer Elementfunktionen keine Definition besitzt. Eine Unterklasse wird konkret, wenn sie alle Eigenschaften definiert, die in der abstrakten Basisklasse offengelassen wurden.

- Eine Elementfunktion, die keine Implementierung hat, wird als rein virtuell bezeichnet. Rein virtuelle Funktionen werden mit »=0« am Ende ihrer Deklaration bezeichnet. Rein virtuelle Funktionen haben keine Definition.
- Eine Klasse, die eine oder mehrere rein virtuelle Funktionen enthält, wird als abstrakte Klasse bezeichnet.
- Eine abstrakte Klasse kann nicht instanziiert werden.
- Eine abstrakte Klasse kann Basisklasse anderer Klassen sein.
- Unterklassen einer abstrakten Klasse werden konkret (d.h. nicht abstrakt), wenn sie alle rein virtuellen Funktionen überschrieben haben, die sie erben.

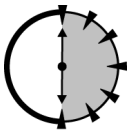
Selbsttest

1. Was ist Faktorieren? (Siehe »Faktorieren«)
2. Was ist das unterscheidende Merkmal einer abstrakten Klasse in C++? (Siehe »Deklaration einer abstrakten Klasse«)
3. Wie erzeugen Sie aus einer abstrakten Klasse eine konkrete Klasse? (Siehe »Erzeugen einer konkreten Klasse aus einer abstrakten Klasse«)
4. Warum ist es möglich, eine Funktion `fn(MyClass*)` zu deklarieren, wenn `MyClass` abstrakt ist? (Siehe »Ein abstraktes Objekt an eine Funktion übergeben«)

Mehrfachvererbung

Checkliste

- ☑ Mehrfachvererbung einführen
- ☑ Uneindeutigkeiten bei Mehrfachvererbung vermeiden
- ☑ Uneindeutigkeiten bei virtueller Vererbung vermeiden
- ☑ Ordnungsregeln für mehrere Konstruktoren wiederholen



30 Min.

In den bisher diskutierten Klassenhierarchien hat jede Klasse von einer einzelnen Elternklasse geerbt. Das ist die Art, wie es auch normalerweise in der realen Welt zugeht. Eine Mikrowelle ist ein Typ Ofen. Man könnte argumentieren, dass eine Mikrowellen Gemeinsamkeiten mit einem Radar hat, der auch Mikrowellen verwendet, aber das ist wirklich ein bißchen weit hergeholt.

Einige Klassen jedoch stellen die Vereinigung zweier Klassen dar. Ein Beispiel einer solchen Klasse ist das Schlafsofa. Wie der Name bereits impliziert, ist es ein Sofa und auch ein Bett (wenn auch kein sehr komfortables). Somit sollte das Schlafsofa Eigenschaften eines Bettes und Eigenschaften eines Sofas erben. Um dieser Situation zu begegnen, erlaubt es C++, eine Klasse von mehr als einer Basisklasse abzuleiten. Das wird Mehrfachvererbung genannt.

24.1 Wie funktioniert Mehrfachvererbung?

Lassen Sie uns das Beispiel mir dem Schlafsofa ausbauen, um die Prinzipien der Mehrfachvererbung zu untersuchen. Abbildung 24.1 zeigt den Vererbungsgraph der Klasse `SleeperSofa`. Beachten Sie, wie die Klasse von der Klasse `Sofa` und der Klasse `Bed` erbt. Auf diese Weise erbt sie die Eigenschaften der beiden Klassen.

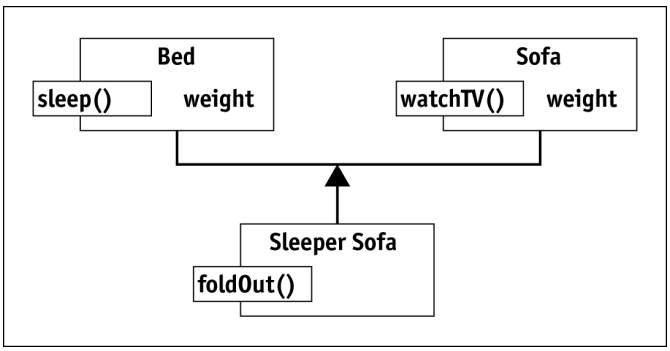


Abbildung 24.1: Klassenhierarchie eines Schlafsofas.

Der Code zur Implementierung von `SleeperSofa` sieht wie folgt aus:

```
// SleeperSofa - demonstriert, wie ein Schlafsofa
//               funktionieren könnte
#include <stdio.h>
#include <iostream.h>

class Bed
{
  public:
    Bed()
    {
        cout << »Teil Bett\n«;
    }
    void sleep()
    {
        cout << »Versuche zu schlafen\n«;
    }
    int weight;
};

class Sofa
{
  public:
    Sofa()
    {
        cout << »Teil Sofa\n«;
    }
    void watchTV()
    {
        cout << »Sehe fern\n«;
    }
    int weight;
};

// SleeperSofa - ist Bett und Sofa
class SleeperSofa : public Bed, public Sofa
{
  public:
```

```
    // der Konstruktor muss nichts machen
    SleeperSofa()
    {
        cout << »Zusammenfügen der beiden\n«;
    }
    void foldOut()
    {
        cout << »Klappe das Bett aus\n«;
    }
};

int main()
{
   SleeperSofa ss;
   // sie können auf einem Schlafsofa fernsehen ...
   ss.watchTV();        // Sofa::watchTV()
   //... und Sie können es ausklappen ...
   ss.foldOut();        // SleeperSofa::foldOut()
   //... und darauf schlafen (irgendwie)
   ss.sleep();          // Bed::sleep()
   return 0;
}
```

Die Namen der beiden Klassen – `Bed` und `Sofa` – kommen nach dem Namen `SleeperSofa`, was anzeigt, dass `SleeperSofa` die Elemente der beiden Basisklassen erbt. Somit sind beide Aufrufe `ss.sleep( )` und `ss.watchTV( )` gültig. Sie können ein `SleeperSofa` entweder als `Bed` oder als `Sofa` benutzen. Zusätzlich kann die Klasse `SleeperSofa` eigene Elemente haben, wie z.B. `foldOut( )`.

Die Ausführung des Programms liefert die folgende Ausgabe:

```
Teil Bett
Teil Sofa
Zusammenfügen der beiden
Sieh fern
Klappe das Bett aus
Versuche zu schlafen
```

Der Teil Bett des Schlafsofas wird zuerst konstruiert, weil die Klasse `Bed` zuerst in der Klassenliste steht, von denen `SleeperSofa` erbt (es hängt nicht an der Reihenfolge, in der die Klassen definiert sind). Danach wird der Teil Sofa des Schlafsofas konstruiert. Schließlich legt `SleeperSofa` selber los.

Nachdem ein `SleeperSofa`-Objekt erzeugt wurde, greift `main( )` nacheinander auf die Elementfunktionen zu – erst wird ferngesehen auf dem Sofa, dann wird das Sofa umgebaut, und dann wird auf dem Sofa geschlafen. (Offensichtlich hätten die Elementfunktionen in jeder Reihenfolge aufgerufen werden können.)

24.2 Uneindeutigkeiten bei Vererbung

Obwohl Mehrfachvererbung ein mächtiges Feature ist, bringt sie doch einige mögliche Probleme für den Programmierer mit sich. Eines ist im vorangegangenen Beispiel zu sehen. Beachten Sie, dass beide Klassen, `Bed` und `Sofa`, ein Element `weight` (Gewicht) enthalten. Das ist logisch, weil beide ein messbares Gewicht haben. Die Frage ist, welches weight von `SleeperSofa` geerbt wird.

Die Antwort ist »beide«. `SleeperSofa` erbt ein Element `Bed::weight` und ein Element `Sofa::weight`. Weil sie den gleichen Namen haben, sind unqualifizierte Referenzen uneindeutig. Der folgende Schnipsel demonstriert das Prinzip:

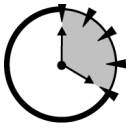
```
int main()
{
   // gib das Gewicht eines Schlafsofas aus
   SleeperSofa ss;
   cout << »Gewicht des Schlafsofas = »
        << ss.weight    // das funktioniert nicht!
        << »\n«;
   return 0;
}
```

Das Programm muss eine der beiden Gewichtsangaben über den entsprechenden Namen der Basisklasse ansprechen. Der folgende Codeschnipsel ist korrekt:

```
#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    cout << »Gewicht des Schlafsofas = »
         << ss.Sofa::weight // Angabe, welches weight
         << »\n«;
}
```

Obwohl diese Lösung das Problem löst, ist die Angabe einer Basisklasse in einer Anwendungsfunktion nicht wünschenswert, weil dadurch Informationen über die Klasse in den Anwendungscode verlagert werden. In diesem Fall muss `fn( )` wissen, dass `SleeperSofa` von `Sofa` erbt.

Diese Typen sogenannter Kollisionen können bei einfacher Vererbung nicht auftreten, sind aber bei Mehrfachvererbung eine ständige Gefahr.



20 Min.

24.3 Virtuelle Vererbung

Im Falle von `SleeperSofa`, war die Kollision der Elemente `weight` mehr, als nur ein Unfall. Ein `SleeperSofa` hat kein Bettgewicht, unabhängig von seinem Sofagewicht – es hat nur ein Gewicht. Die Kollision entsteht, weil diese Klassenhierarchie die reale Welt nicht vollständig beschreibt. Insbesondere wurden die Klassen nicht vollständig faktoriert.

Wenn man etwas mehr nachdenkt, wird klar, dass Betten und Sofas Spezialfälle eines grundlegenderen Konzeptes sind: Möbel. (Natürlich könnte ich das Konzept noch viel fundamentaler machen z.B. mit einer Klasse `ObjectWithMass` (=Objekte mit Masse), aber `Furniture` (=Möbel) ist fundamental genug.) Gewicht ist eine Eigenschaft von allen Möbelstücken. Diese Beziehung ist in Abbildung 24.2 zu sehen:

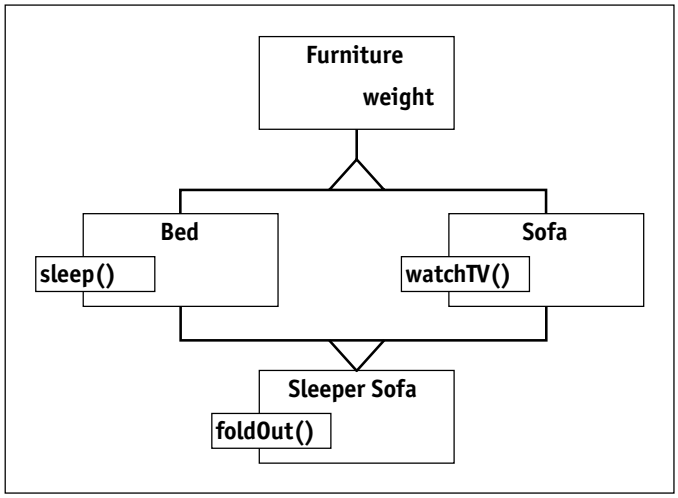


Abbildung 24.2: Weiteres Faktorieren von Bed und Sofa.

Die Klasse `Furniture` zu faktorieren sollte die Namenkollision auflösen. Sehr erleichtert, und mit großer Hoffnung auf Erfolg, realisiere ich die folgende C++-Hierarchie im Programm AmbiguousInheritance:

```
// AmbiguousInheritance - beide Klassen Bed und Sofa
//                        können von einer Klasse
//                        Furniture erben
#include <stdio.h>
#include <iostream.h>

class Furniture
{
  public:
    Furniture()
    {
        cout << »Erzeugen des Konzeptes Furniture«;
    }
    int weight;
};

class Bed : public Furniture
{
  public:
    Bed()
    {
        cout << »Teil Bett\n«;
    }
    void sleep()
    {
        cout << »Versuche zu schlafen\n«;
    }
    int weight;
};
```

```
class Sofa : public Furniture
{
  public:
    Sofa()
    {
        cout << »Teil Sofa\n«;
    }
    void watchTV()
    {
        cout << »Sehe fern\n«;
    }
    int weight;
};

// SleeperSofa - ist Bett und Sofa
class SleeperSofa : public Bed, public Sofa
{
  public:
    // der Konstruktor muss nichts machen
    SleeperSofa()
    {
        cout << »Zusammenfügen der beiden\n«;
    }
    void foldOut()
    {
        cout << »Klappe das Bett aus\n«;
    }
};

int main()
{
   // Ausgabe des Gewichts eines Schlafsofas
   SleeperSofa ss;
   cout << »Gewicht des Schlafsofas = »
        << ss.weight    // das funktioniert nicht!
        << »\n«;
   return 0;
}
```

Unglücklicherweise hilft das gar nicht – weight ist immer noch uneindeutig. »OK«, sage ich (wobei ich nicht wirklich verstehe, warum `weight` immer noch uneindeutig ist), »Ich werde es einfach nach `Furniture` casten«.

```
#include <iostream.h>
void fn()
{
    SleeperSofa ss;
    Furniture *pF;
    pF = (Furniture*)&ss; // nutze Zeiger auf
                          // Furniture...
   cout << »weight = »    //... um an das Gewicht
        << pF->weight     // zu kommen
        << »\n«;
};
```

Auch das funktioniert nicht. Jetzt bekomme ich eine Fehlermeldung, dass der Cast von `SleeperSofa*` nach `Furniture*` uneindeutig ist. Was geht hier eigentlich vor?

Die Erklärung ist einfach. `SleeperSofa` erbt nicht direkt von Furniture. Beide Klassen, `Bed` und `Sofa`, erben von `Furniture`, und `SleeperSofa` erbt dann von beiden. Im Speicher sieht ein `SleeperSofa-Objekt` wie in Abbildung 24.3 aus:

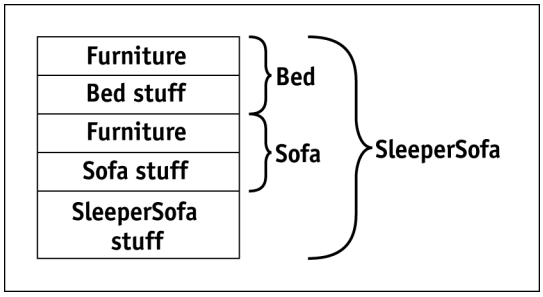


Abbildung 24.3: Speicheranordnung eines SleeperSofa.

Sie sehen, dass `SleeperSofa` aus einem vollständigen `Bed` besteht, gefolgt von einem vollständigen `Sofa`, gefolgt von Dingen, die für `SleeperSofa` spezifisch sind. Jedes dieser Teilobjekte in `SleeperSofa` hat seinen eigenen `Furniture`-Teil, weil jeder von `Furniture` erbt. Somit enthält ein `SleeperSofa` zwei `Furniture`-Objekte.

Ich habe nicht die Hierarchie in Abbildung 24.2 erzeugt, sondern eine Hierarchie, wie sie in Abbildung 24.4 zu sehen ist.

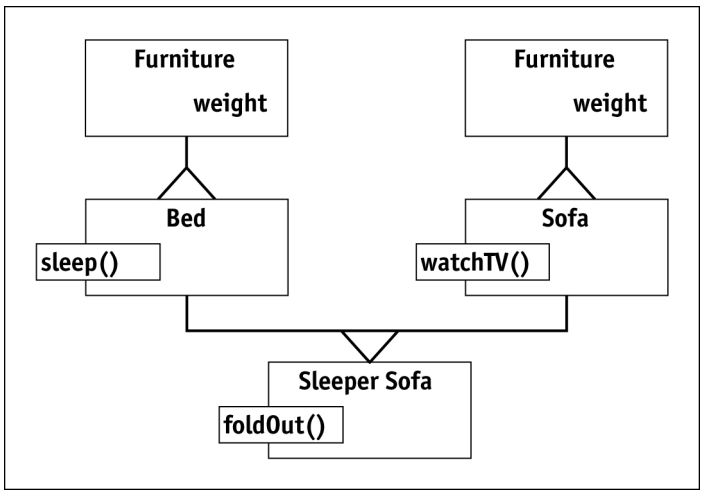


Abbildung 24.4: Tatsächliches Ergebnis des ersten Faktorierens von Bed und Sofa.

Das ist aber Unsinn. `SlepperSofa` braucht nur eine Kopie von `Furniture`. Ich möchte, dass `SleeperSofa` nur eine Kopie von `Furniture` erbt, somit möchte ich, dass `Bed` und `Sofa` sich diese eine Kopie teilen.

C++ nennt das *virtuelle Vererbung*, weil sie das Schlüsselwort `virtual` verwendet.

Ich mag dieses Überladen des Begriffs virtual nicht, weil virtuelle Vererbung nichts mit virtuellen Funktionen zu tun hat.

Ich kehre zur Klasse `SleeperSofa` zurück, und implementiere sie wie folgt:

```
// MultipleVirtual - basiere SleeperSofa auf einer
//                   einzelnen Kopie von Furniture
// Dieses Programm kann kompiliert werden!
#include <stdio.h>
#include <iostream.h>

class Furniture
{
  public:
    Furniture()
    {
        cout << »Erzeugen des Konzeptes Furniture«;
    }
    int weight;
};

class Bed : virtual public Furniture
{
  public:
    Bed()
    {
        cout << »Teil Bett\n«;
    }
    void sleep()
    {
        cout << »Versuche zu schlafen\n«;
    }
    int weight;
};

class Sofa : virtual public Furniture
{
  public:
    Sofa()
    {
        cout << »Teil Sofa\n«;
    }
    void watchTV()
    {
        cout << »Sehe fern\n«;
    }
    int weight;
};

// SleeperSofa - ist Bett und Sofa
```

```
class SleeperSofa : public Bed, public Sofa
{
  public:
    // der Konstruktor muss nichts machen
    SleeperSofa()
    {
        cout << »Zusammenfügen der beiden\n«;
    }
    void foldOut()
    {
        cout << »Klappe das Bett aus\n«;
    }
};

int main()
{
   // Ausgabe des Gewichts eines Schlafsofas
   SleeperSofa ss;
   cout << »Gewicht des Schlafsofas = «
        << ss.weight    // das funktioniert!
        << »\n«;
   return 0;
}
```

Beachten Sie, dass das Schlüsselwort `virtual` bei der Vererbung von `Furniture` in `Bed` und `Sofa` eingefügt wurde. Das drückt aus »Gib mir eine Kopie von `Furniture`, wenn noch keine solche vorhanden ist, ansonsten verwende diese.« Ein `SleeperSofa` sieht im Speicher schließlich so aus:

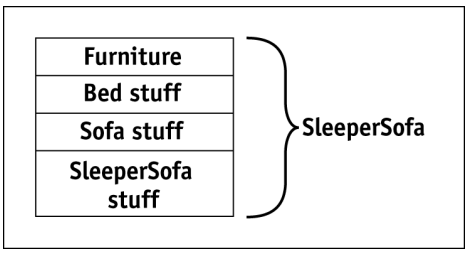


Abbildung 24.5: Speicheranordnung von SleeperSofa bei virtueller Vererbung.

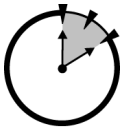
Ein `SleeperSofa` erbt von `Furniture`, dann von `Bed` minus `Funiture`, gefolgt von `Sofa` minus `Furniture`. Dadurch werden die Elemente in `SleeperSofa` eindeutig. (Das muss nicht die Reihenfolge der Elemente im Speicher sein, das ist aber für unsere Zwecke auch nicht wichtig.)

Die Referenz in `main( )` auf `weight` ist nicht länger uneindeutig, weil ein `SleeperSofa` nur eine Kopie von `Furniture` enthält. Indem von `Furniture` virtuell geerbt wird, bekommen sie die Vererbungsbeziehung wie in Abbildung 24.2.

Wenn virtuelle Vererbung das Problem so schön löst, warum wird sie dann nicht immer verwendet? Dafür gibt es zwei Gründe. Erstens werden virtuell geerbte Basisklassen intern sehr verschieden von normal geerbten Klassen behandelt und diese Unterschiede schließen einen Mehraufwand ein. (Kein sehr großer Mehraufwand, aber die Erfinder von C++ waren fast paranoid, wenn es um Mehraufwand ging.) Zweitens möchten Sie manchmal zwei Kopien der Basisklasse haben (obwohl das unüblich ist).

Ich denke, virtuelle Vererbung sollte die Regel sein.

Als ein Beispiel für einen Fall, in dem sie keine virtuelle Vererbung haben möchten, betrachten Sie das Beispiel `TeacherAssistent`, der gleichzeitig `Student` und `Teacher` ist; beide Klassen sind Unterklassen von `Academician`. Wenn die Universität jedem TeachingAssistent zwei IDs gibt – eine Student-ID und eine Teacher-ID – muss die Klasse `TeacherAssistant` zwei Kopien der Klasse `Academician` enthalten.



10 Min.

24.4 Konstruktion von Objekten bei Mehrfachnennung

Die Regeln für die Konstruktion von Objekten müssen auf die Behandlung von Mehrfachvererbung ausgedehnt werden. Die Konstruktoren werden in dieser Reihenfolge aufgerufen:

- Zuerst wird der Konstruktor jeder virtuellen Basisklasse aufgerufen, in der Reihenfolge, in der von den Klassen geerbt wird.
- Dann wird der Konstruktor jeder nicht virtuellen Basisklasse aufgerufen, in der Reihenfolge, in der von den Klassen geerbt wird.
- Dann wird der Konstruktor aller Elementobjekte aufgerufen, in der Reihenfolge, in der die Elementobjekte in der Klasse erscheinen.
- Schließlich wird der Konstruktor der Klasse selber aufgerufen.
- Basisklassen werden in der Reihenfolge konstruiert, in der von ihnen geerbt wird, und nicht in der Reihenfolge in der Konstruktorzeile.

24.5 Eine Meinung dagegen

Nicht alle objektorientierten Praktiker sind der Meinung, dass Mehrfachvererbung eine gute Idee ist. Außerdem unterstützen nicht alle objektorientierten Sprachen Mehrfachvererbung. Java z.B. unterstützt keine Mehrfachvererbung – diese wird als zu gefährlich eingeschätzt und ist den damit verbundenen Ärger nicht wert.

Mehrfachvererbung ist für die Sprache nicht leicht zu implementieren.

Das ist hauptsächlich ein Compilerproblem (oder ein Problem des Compilerschreibers) und nicht das Problem des Programmierers. Mehrfachvererbung öffnet jedoch die Tür für neue Fehler. Erstens gibt es Uneindeutigkeiten, die im Abschnitt »Uneindeutigkeit bei Vererbung« erwähnt wurden. Zweitens schließt in Anwesenheit von Mehrfachvererbung das Casten eines Zeigers auf eine Unterklasse in einen Zeiger auf eine Basisklasse mysteriöse Konvertierungen ein, die unerwartete Resultate liefern können. Hier ein Beispiel:

```
#include <iostream.h>
class Base1 {int mem;};
class Base2 {int mem;};
class SubClass : public Base1, public Base2 {};

void fn(SubClass *pSC)
{
   Base1 *pB1 = (Base1*)pSC;
   Base2 *pB2 = (Base2*)pSC;
   if ((void*)pB1 == (void*)pB2)
   {
      cout << »Elemente nummerisch gleich\n«;
   }
}
int main()
{
   SubClass sc;
   fn(&sc);
   return 0;
}
```

`pB1` und `pB2` sind nummerisch nicht gleich, obwohl sie vom selben Originalwert `pSC` kommen. Aber die Meldung »Elemente nummerisch gleich« kommt nicht. (Tatsächlich wird `fn( )` eine Null übergeben, weil C++ diese Konvertierungen auf `null` nicht ausführt. Sehen Sie, wie merkwürdig das werden kann?)

0 Min.

Zusammenfassung

Ich empfehle Ihnen, Mehrfachvererbung nicht zu benutzen, bis sie C++ gut beherrschen. Einfache Vererbung stellt bereits genügend Ausdrucksstärke bereit, die genutzt werden kann. Später können sie die Handbücher studieren, bis Sie ganz sicher sind, dass Sie verstehen, was passiert, wenn Sie Mehrfachvererbung verwenden. Eine Ausnahme ist die Verwendung der Microsoft Foundation Classes (MFC), die Mehrfachvererbung nutzen. Diese Klassen wurden getestet und sind sicher. (Sie werden im Allgemeinen nicht einmal merken, dass Bibliotheken wie MFC Mehrfachvererbung nutzen.)

- Eine Klasse kann von mehr als einer Klasse erben, indem deren Klassennamen durch Kommata getrennt hinter dem »:« stehen. Obwohl in den Beispielen nur zwei Basisklassen verwendet wurden, gibt es keine Beschränkung für die Anzahl der Basisklassen. Vererbung von mehr als zwei Basisklassen ist jedoch sehr ungewöhnlich.
- Elemente, die in den Basisklassen gleich sind, sind in der Unterklasse uneindeutig. D.h. wenn beide, `BaseClass1` und `BaseClass2` eine Elementfunktion `f( )` enthalten, dann ist `f( )` uneindeutig in `SubClass`.
- Uneindeutigkeiten in den Basisklassen können über einen Klassenanzeiger aufgelöst werden, d.h. eine Unterklasse kann sich auf `BaseClass1::f( )` und `BaseClass2::f( )` beziehen.
- Wenn beide Basisklassen von einer gemeinsamen Basisklasse abgeleitet sind, in der gemeinsame Eigenschaften faktoriert sind, kann das Problem mit virtueller Vererbung gelöst werden.

Selbsttest

1. Was könnten wir als Basisklassen für eine Klasse wie `CombinationPrinterCopier` benutzen? (Ein Druck-Kopierer ist ein Laserdrucker, der auch als Kopierer verwendet werden kann.) (Siehe Einleitungsabschnitt)
2. Vervollständigen Sie die folgende Klassenbeschreibung, indem Sie die Fragezeichen ersetzen:

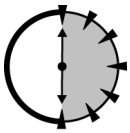
```
class Printer
{
  public:
    int nVoltage;
    // ... weitere Elemente ...
}
class Copier
{
  public:
    int nVolatage;
    // ... weitere Elemente ...
}
class CombinationPinterCopier ?????
{
    // .... Weiteres ...
}
```

3. Was ist das Hauptproblem beim Zugriff auf voltage eines `CombinationPrinterCopier-Objektes`? (Siehe »Uneindeutigkeiten bei Vererbung«)
4. Gegeben, dass beide, `Printer` und `Copier`, `ElectronicEquipment` sind, was kann getan werden, um das voltage-Problem zu lösen? (Siehe »Virtuelle Vererbung«)
5. Nennen Sie einige Gründe, warum Mehrfachvererbung keine gute Sache sein kann. (Siehe »Eine Meinung dagegen«)

Große Programme

Checkliste

- ☑ Programme in mehrere Module teilen
- ☑ Die `#include`-Direktive verwenden
- ☑ Dateien einem Projekt hinzufügen
- ☑ Andere Kommandos des Präprozessors



30 Min.

Alle bisherigen Programme waren klein genug, um sie in eine einzige .cpp-Datei zu schreiben. Das ist für die Beispiele in einem Buch wie *C++-Wochenend-Crashkurs* in Ordnung, wäre aber für reale Anwendung eine ernsthafte Beschränkung. Diese Sitzung untersucht, wie ein Programm in mehrere Teile aufgeteilt werden kann durch die clevere Verwendung von Projekt- und Include-Dateien.

25.1 Warum Programme aufteilen?

Der Programmierer kann ein Programm in mehrere Dateien aufteilen, die manchmal auch *Module* genannt werden. Diese einzelnen Quelldateien werden separat kompiliert und dann im Erzeugungsprozess zu einem einzigen Programm zusammengefügt.

Der Prozess, separat kompilierte Module zu einer ausführbaren Datei zusammenzufügen, wird als Linken bezeichnet.

Es gibt mehrere Gründe dafür, ein Programm in handlichere Teile zu teilen. Erstens ermöglicht das Teilen in Module eine höhere Kapselung. Klassen mauern ihre Elemente ein, um einen gewissen Grad von Sicherheit zu erreichen. Programme können dasselbe mit Funktionen tun.

Erinnern Sie sich daran, dass Kapselung einer der Vorteile von objektorientierter Programmierung ist.

Zweitens ist ein Programm, das aus einer Anzahl gut durchdachter Module besteht, leichter zu verstehen, und damit leichter zu schreiben und zu debuggen, als ein Programm, das nur eine Quelldatei besitzt, in der alle Klassen und Funktionen enthalten sind.

Dann kommt die Wiederverwendung. Ich habe das Argument der Wiederverwendbarkeit gebraucht, um Ihnen die objektorientierte Programmierung zu verkaufen. Es ist extrem schwierig, eine einzelne Klasse zu pflegen, die in mehreren Programmen verwendet wird, wenn jedes Programm seine eigene Kopie der Klasse enthält. Es ist viel besser, wenn ein einziges Klassenmodul automatisch von den Programmen geteilt wird.

Schließlich gibt es noch ein Zeitargument. Compiler wie Visual C++ oder GNU C++ brauchen nicht sehr lange für das Kompilieren der Beispielprogramme in diesem Buch auf einem so schnellem Rechner wie dem Ihren. Kommerzielle Programme bestehen manchmal aus einigen Millionen Zeilen Quelltext. Ein solches Programm zu erzeugen kann mehr aus 24 Stunden in Anspruch nehmen! (Fast so lange wie sie benötigen, dieses Buch zu lesen!) Kein Programmierer würde es hinnehmen, ein solches Programm wegen jeder kleinen Änderung neu kompilieren zu müssen. Die Zeit zum Kompilieren ist wesentlich länger als die Zeit zum Linken.

25.2 Trennung von Klassendefinition und Anwendungsprogramm

Dieser Abschnitt beginnt mit dem Beispiel EarlyBinding aus Sitzung 22, und trennt die Definition der Klasse `Student` vom Rest des Programms. Um Verwechslungen zu vermeiden, lassen Sie uns das Ergebnis `SeparatedClass` nennen.

25.2.1 Aufteilen des Programms

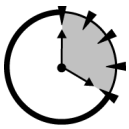
Wir beginnen mit der Aufteilung von SeparatedClass in logische Einheiten. Natürlich können die Anwendungsfunktionen `fn( )` und `main( )` von der Klassendefinition getrennt werden. Diese Funktionen sind weder wiederverwendbar, noch haben sie etwas mit der Definition von Student zu tun. In gleicher Weise hat die Klasse Student keine Beziehung zu den Funktionen `fn( )` oder `main( )`.

Ich speichere den Anwendungsteil des Programms in einer Datei SeparatedClass.cpp. Bis jetzt sieht das Programm so aus:

```
// SeparatedClass - demonstriert eine Anwendung
//                  getrennt von der
//                  Klassendefinition
#include <stdio.h>
#include <iostream.h>

double fn(Student& fs)
```

```
{
    // weil calcTuition() virtual deklariert ist,
    // wird der Laufzeittyp von fs verwendet, um
    // den Aufruf aufzulösen
    return fs.calcTuition();
}

int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // fn() mit einem Student-Objekt
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(s)
         << »\n\n«;

    // der folgende Ausdruck ruft
    // fn() mit einem GraduateStudent-Objekt
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(gs)
         << »\n\n«;
    return 0;
}
```

Unglücklicherweise kann das Modul nicht erfolgreich kompiliert werden, weil nichts in SeparatedClass.cpp die Klasse `Student` definiert. Wir könnten natürlich die Definition von `Student` wieder in die Datei SeparatedClass.cpp einfügen, aber das ist nicht, was wir wollen. Wir würden damit dorthin zurückkehren, wo wir hergekommen sind.



20 Min.

25.2.2 Die #include-Direktive

Was benötigt wird, ist eine Methode, um die Deklaration von Student programmtechnisch in SeparatedClass.cpp einzubinden. Die `#include`-Direktive tut genau das. Die `#include`-Direktive fügt den Inhalt einer Datei, die im Quelltext benannt ist, an Stelle der `#include`-Direktive ein. Das ist schwerer zu erklären, als es in der Praxis ist.

Zuerst erzeuge ich eine Datei `student.h`, die die Definition der Klassen Student und `GraduateStudent` enthält:

```
// Student - definiere Eigenschaften von Student
class Student
{
  public:
    virtual double calcTuition()
    {
        return 0;
    }

  protected:
```

```
    int nID;
};
class GraduateStudent : public Student
{
  public:
    virtual double calcTuition()
    {
        return 1;
    }

  protected:
    int nGradId;
};
```

Die Zieldatei der* `#include`*-Direktive wird auch als Include-Datei bezeichnet. Nach Konvention tragen die Include-Dateien den Namen der Basisklasse, die sie enthalten, in Kleinbuchstaben und mit einer .h-Endung. Sie werden auch C++-Include-Dateien finden mit den Endungen .hh, .hpp und .hxx. Theoretisch kümmert sich der Compiler nicht darum.

Die neue Version der Quelldatei SeparatedClass.cpp unserer Anwendung sieht wie folgt aus:

```
// SeparatedClass - demonstriert eine Anwendung
//                  getrennt von der
//                  Klassendefinition
#include <stdio.h>
#include <iostream.h>

#include »student.h«

double fn(Student& fs)
{
    // ... von hier ab identisch mit
    // voriger Version ...
```

Die `#include`-Direktive wurde hinzugefügt.

Die* `#include`*-Direktive muss in der ersten Spalte beginnen und darf nur eine Zeile umfassen.

Wenn Sie den Inhalt von student.h physikalisch in die Datei SeparatedClass.cpp einfügen, kommen Sie zu der gleichen Datei LateBinding.cpp, mit der wir gestartet sind. Das ist genau das, was während des Erzeugungsprozesses passiert – C++ fügt student.h in SeparatedClass.cpp ein und kompiliert das Ergebnis.

Die `#include`-Direktive hat nicht die gleiche Syntax wie die anderen C++-Kommandos. Das liegt daran, dass es überhaupt keine C++-Direktive ist. Ein Präprozessor geht zuerst über das C++-Programm, bevor der C++-Compiler mit der Ausführung beginnt. Es ist der Präprozessor, der die `#include`-Direktive interpretiert.

25.2.3 Anwendungscode aufteilen

Das Programm SeparatedClass trennt die Klassendefinition erfolgreich vom Anwendungscode, aber nehmen Sie einmal an, das reicht nicht – nehmen Sie an, wir wollten die Funktion `fn( )` von `main( )` trennen. Ich könnte natürlich die gleiche Vorgehensweise anwenden und eine Datei fn.h erzeugen, die vom Hauptprogramm eingebunden wird.

Diese Lösung der Include-Datei löst nicht das Problem mit den Programmen, die ewig für ihre Erzeugung brauchen. Außerdem bringt die Lösung alle Probleme mit sich, die sich darum drehen, welche Funktion welche andere Funktion aufrufen kann, abhängig von den Include-Anweisungen. Eine bessere Lösung ist die Aufteilung des Quellcodes in separate Kompilierungseinheiten.

Während der Kompilierungsphase des Erzeugungsprozesses konvertiert C++ den Quelltext in den .cpp-Dateien in äquivalenten Maschinencode. Dieser Maschinencode wird in einer Datei gespeichert, die als Objektdatei bezeichnet wird, und die Endung .obj (Visual C++) oder .o (GNU C++) trägt. In einer anschließenden Phase, die als Linkphase bezeichnet wird, werden die Objektdateien mit der C++-Standardbibliothek zusammengefügt, um das ausführbare Programm zu bilden.

Lassen Sie uns diese Fähigkeiten zu unserem Vorteil nutzen. Wir können die Datei SeparatedClass.cpp in eine Datei SeparatedFn.cpp und eine Datei SeparatedMain.cpp aufteilen. Wir beginnen mit dem Anlegen dieser beiden Dateien.

Die Datei SeparatedFn.cpp sieht wie folgt aus:

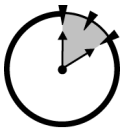
```
// SeparatedFn - demonstriert eine Anwendung, die in
//               zwei Teile geteilt ist - der Teil
//               von fn()
#include <stdio.h>
#include <iostream.h>

#include »student.h«

double fn(Student& fs)
{
    // weil calcTuition() virtual deklariert ist,
    // wird der Laufzeittyp von fs verwendet, um
    // den Aufruf aufzulösen
    return fs.calcTuition();
}
```

Der Rest des Programms, die Datei SeparatedMain.cpp, sieht so aus:

```
// SeparatedMain - demonstriert eine Anwendung, die
//                 in zwei Teile geteilt ist -
//                 der Teil von main( )
#include <stdio.h>
#include <iostream.h>

#include »student.h«
```

```
int main(int nArgc, char* pszArgs[])
{
    // der folgende Ausdruck ruft
    // fn() mit einem Student-Objekt
    Student s;
    cout << »Der Wert von s.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(s)
         << »\n\n«;

    // der folgende Ausdruck ruft
    // fn() mit einem GraduateStudent-Objekt
    GraduateStudent gs;
    cout << »Der Wert von gs.calcTuition bei\n«
         << »virtuellem Aufruf durch fn() ist »
         << fn(gs)
         << »\n\n«;
    return 0;
}
```

Beide Quelldateien binden die gleichen .h-Dateien ein, weil beide Zugriff auf die Definition der Klasse `Student` und der Funktionen in der C++-Standardbibliothek benötigen.



10 Min.

25.2.4 Projektdatei

Voller Erwartung öffne ich die Datei SeparatedMain.cpp und wähle »`Build`« aus.

Wenn Sie das zu Hause versuchen. stellen Sie sicher, dass sie die Projektdatei SeparatedClass geschlossen haben.

Eine Fehlermeldung »undeclared identifier« erscheint. C++ weiß nicht, was `fn( )` ist, wenn SeparatedMain.cpp kompiliert wird. Das macht Sinn, weil die Definition von `fn( )` in einer anderen Datei steht.

Natürlich muss ich eine Prototypdeklaration von `fn( )` in die Quelldatei SeparatedMain.cpp einfügen:

```
double fn(Student& fs);
```

Die resultierende Quelldatei lässt sich kompilieren, erzeugt aber während des Linkens einen Fehler, dass die Funktion `fn(Student)` in den .o-Dateien nicht gefunden werden kann.

Ich könnte (und sollte wahrscheinlich auch) einen Prototyp von `main( )` ***in die Datei SeparatedFn.cpp einfügen; das ist jedoch nicht nötig, weil*** `fn( )` ***die Funktion*** `main( )` ***nicht aufruft.***

Was benötigt wird, ist eine Möglichkeit, C++ mitzuteilen, beide Quelldateien im gleichen Programm zusammenzubinden. Solch eine Datei wird Projektdatei genannt. Es gibt mehrere Wege, wie eine Projektdatei angelegt werden kann. Die Techniken unterscheiden sich in den zwei Compilern.

Erzeugen einer Projektdatei unter Visual C++

Für Visual C++ führen Sie die folgenden Schritte aus:

1. Stellen Sie sicher, dass Sie andere Projektdateien, die von früheren Versuchen stammen, geschlossen haben, indem Sie auf »Arbeitsbereich schließen« im Datei-Menü klicken. (Ein Arbeitsbereich ist der Name, den Microsoft für eine Sammlung von Projektdateien verwendet.)
2. Öffnen Sie die Quelldatei SeparatedMain.cpp. Klicken Sie auf »Compile«.
3. Wenn Visual C++ sie fragt, ob sie eine Projektdatei erzeugen möchten, antworten Sie mit Ja. Sie haben nun eine Projektdatei, die die einzelne Datei SeparatedMain.cpp enthält.
4. Wenn noch nicht geöffnet, öffnen sie das Workspace-Fenster (Workspace unterhalb von View klicken).
5. Schalten Sie auf FileView um. Klicken Sie auf SeparatedMain files, wie in Abbildung 25.1 zu sehen ist. Wählen Sie Add Files to Projekt aus. Vom Menü aus öffnen Sie die Quelldatei SeparatedFn.cpp. Beide Dateien SeparatedMain.cpp und SeparatedFn.cpp erscheinen nun in der Liste der Dateien, die zu dem Projekt gehören.
6. Klicken Sie Build, um das Programm erfolgreich zu erzeugen. (Das erste Mal werden beide Quelldateien kompiliert, wenn Sie Build All ausführen.)

Ich habe nicht behauptet, dies sei der eleganteste Weg. Aber es ist der einfachste.

Abbildung 25.1: Klicken Sie auf den rechten Mausbutton, um Dateien in das Projekt einzufügen.

Die Projektdatei SeparatedMain auf der beiliegenden CD-ROM enthält bereits beide Quelldateien.

Erzeugen einer Projektdatei unter GNU C++

Verwenden Sie die folgenden Schritte, um unter `rhide`, der GNU C++-Umgebung, eine Projektdatei zu erzeugen.

1. Ohne eine Datei offen zu haben, klicken Sie auf »Projekt öffnen« in Projekt-Menü.
2. Schreiben Sie den Namen SeparatedMainGNU.pr (der Name ist nicht wichtig – sie können ihn wählen wie sie möchten). Ein Projektfenster mit einem einzigen Eintrag, <empty>, wird am unteren Rand des Fensters geöffnet.
3. Clicken Sie auf »Add Item« unter Projekt. Öffnen Sie die Datei SeparatedMain.cpp.
4. Verfahren Sie mit SeparatedFn.cpp ebenso.
5. Wählen Sie »Make« im Compile-Menü aus, um das Programm SeparatedMainGNU.exe erfolgreich zu erzeugen. (Make erzeugt nur diese Dateien neu, die geändert wurden; »Neu erzeugen« erzeugt alle Dateien neu, ob sie geändert wurden oder nicht.)

Abbildung 25.2 zeigt den Inhalt des Projektfensters zusammen mit dem Meldungsfenster, das während des Erzeugungsprozesses gezeigt wird.

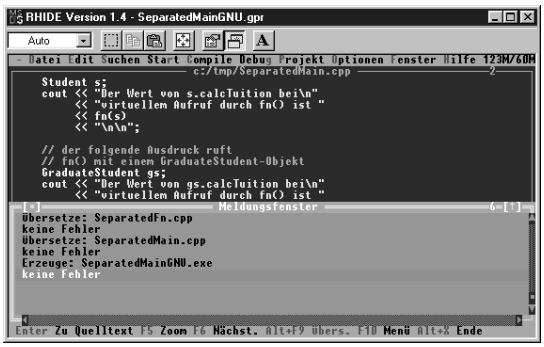


Abbildung 25.2: Die rhide-Umgebung zeigt die kompilierten Dateien und das erzeugte Programm an.

25.2.5 Erneute Betrachtung des Standard-Programm-Templates

Jetzt können Sie sehen, warum wir die Direktiven `#include <stio.h>` und `#include <iostream.h>` in unseren Programmen verwendet haben. Diese Include-Dateien enthalten die Definitionen der Funktionen und Klassen, die verwendet wurden, wie z.B. `strcat( )` und `cin`.

Die von Standard-C++ definierten .h-Dateien werden durch die Klammern `<>` eingebunden, während lokal definierte .h-Dateien durch Hochkommata definiert werden. Der einzige Unterschied zwischen den beiden Schreibweisen ist, dass C++ die in Hochkommata eingeschlossenen Dateien zuerst im aktuellen Verzeichnis sucht (das Verzeichnis, das die Projektdatei enthält) und C++ die Suche für Dateinamen in Klammern im Verzeichnis der C++-Include-Dateien beginnt. Für beide Wege kann der Programmierer die zu durchsuchenden Verzeichnisse durch Einstellungen in der Projektdatei beeinflussen.

In der Tat ist es das Konzept des separaten Kompilierens, das Include-Dateien kritisch macht. Beide, SeparatedFn und SeparatedMain kennen `Student`, weil student.h eingebunden wurde. Wir hätten auch diese Definition in beide Quelldateien hineinschreiben können, das wäre aber sehr gefährlich gewesen. Die gleiche Definition an zwei verschiedenen Stellen schafft die Möglichkeit, dass die

beiden nicht synchronisiert werden – eine Definition könnte geändert werden und die andere nicht.

Die Definition von `Student` in eine einzige Datei zu schreiben und diese Datei in zwei Module einzubinden, macht die Entstehung verschiedener Definitionen unmöglich.

25.2.6 Handhabung von Outline-Elementfunktionen

Die Beispielklassen `Student` und `GraduateStudent` definieren ihre Funktionen innerhalb der Klasse; die Elementfunktionen sollten aber außerhalb der Klasse deklariert sein (nur die Klasse `Student` wird gezeigt – die Klasse `GraduateStudent` ist identisch).

```
// Student - definierte Eigenschaften von Student
class Student
{
  public:
    // deklariere Elementfunktion
    virtual double calcTuition();

  protected:
    int nID;
};

// definiere den Code separat von der Klasse
double Student::calcTuition();
{
    return 0;
}
```

Ein Problem tritt auf, wenn der Programmierer beide Dateien, die Klasse und die Elementfunktionen, in die gleiche .h-Datei einzubinden versucht. Die Funktion `Student::calcTuition( )` wird ein Teil von SeparatedMain.o und SeparatedFn.o. Wenn diese Dateien gelinkt werden, wird sich der C++-Linker darüber beschweren, dass `calcTuition( )` zweimal definiert ist.

Wenn eine Elementfunktion innerhalb einer Klasse definiert ist, unternimmt C++ gewisse Anstrengungen, Doppeldefinitionen von Funktionen zu vermeiden. C++ kann dieses Problem nicht verhindern, wenn die Elementfunktion außerhalb der Klasse definiert ist.

Externe Elementfunktionen müssen in ihrer eigenen .cpp-Datei definiert werden, wie in der folgenden Datei Student.cpp:

```
#include »student.h«
// definiere den Code separat von der Klasse
double Student::calcTuition();
{
    return 0;
}
```

0 Min.

Zusammenfassung

Diese Sitzung hat Ihnen gezeigt, wie der Programmierer Programme in mehrere Quelldateien aufspalten kann. Kleinere Quelldateien sparen Zeit zur Programmgenerierung, weil der Programmierer nur die Sourcemodule kompilieren muss, die tatsächlich geändert wurden.

- Separat kompilierte Module steigern die Kapselung von Paketen ähnlicher Funktionen. Wie Sie bereits gesehen haben, sind separate, gekapselte Pakete, einfacher zu schreiben und zu debuggen. Die C++-Standardbibliothek ist ein solches gekapseltes Paket.
- Der Generierungsprozess besteht aus zwei Phasen. In der ersten, der Kompilierungsphase, werden die C++-Anweisungen in maschinenlesbare, aber unvollständige Objektdateien übersetzt. In der zweiten Phase, der Linkphase, werden diese Objektdateien zu einer einzigen ausführbaren Datei zusammengefügt.
- Deklarationen, Klassendefinitionen eingeschlossen, müssen zusammen mit jeder C++-Quelldatei kompiliert werden, die diese Funktion oder Klasse deklariert. Der einfachste Weg, dies zu bewerkstelligen, ist der verwandte Deklarationen in eine .h-Datei zu schreiben, die dann in den .cpp-Quelldateien mittels einer `#include`-Direktive eingebunden wird.
- Die Projektdatei listet die Module auf, die das Programm bilden. Die Projektdatei enthält weitere programmspezifische Einstellungen, die Einfluss darauf haben, wie die C++-Umgebung das Programm erzeugt.

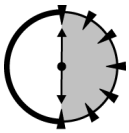
Selbsttest

1. Wie wird eine C++-Quelldatei in eine maschinenlesbare Objektdatei überführt? Wie nennt man diesen Vorgang? (Siehe »Warum Programme aufteilen?«)
2. Wie nennt man den Prozess, der diese Objektdateien zu einer einzigen ausführbaren Datei zusammenfügt? (Siehe »Warum Programme aufteilen?«)
3. Welche Aufgaben hat die Projektdatei? (Siehe »Projektdatei«)
4. Was ist die Hauptaufgabe der `#include`-Direktive? (Siehe »Die #include-Direktive«)

C++-Präprozessor

Checkliste

- ☑ Häufig benutzte Konstanten mit Namen versehen
- ☑ Compilezeitmakros definieren
- ☑ Den Kompilierungsprozess kontrollieren



30 Min.

Die Programme in Sitzung 25 nutzten die #include-Direktive des Präprozessors, um die Definition von Klassen in mehrere Quelldateien einzubinden, die gemeinsam das Programm bildeten. In der Tat haben alle bisher gesehenen Programme stdio.h und iostream.h eingebunden, in denen Funktionen der Standardbibliothek definiert sind. In dieser Sitzung untersuchen wir die #include-Direktive in Verbindung mit anderen Präprozessorkommandos.

26.1 Der C++-Präprozessor

Als C++-Programmierer klicken Sie und ich auf das Build-Kommando, um den C++-Compiler anzuweisen, unseren Quellcode in ein ausführbares Programm zu übersetzen. Wir kümmern uns in der Regel nicht um die Details, wie das Kompilieren abläuft. In Sitzung 25 haben Sie gelernt, dass der Erzeugungsprozess aus zwei Teilen besteht, einer Kompilierungsphase, die jede .cpp-Datei in Maschinencode übersetzt, und einer Linkphase, die diese Objektdateien zusammen mit den Bibliotheksdateien von C++ zu einer ausführbaren .exe-Datei zusammenfasst. Was noch nicht klar wurde, ist, dass die Compilierungsphase selber aus verschiedenen Phasen besteht.

Der Compiler operiert auf Ihren C++-Quelldateien in mehreren Schritten. Der erste Schritt findet und identifiziert alle Variablen und Klassendeklarationen, während ein weiterer Schritt den Objektcode generiert. Jeder Compiler macht so viele oder wenige Schritte wie er braucht – es gibt dafür keinen C++-Standard.

Noch vor dem ersten Compilerschritt bekommt jedoch der C++-Präprozessor eine Chance. Der C++-Präprozessor geht durch die .cpp-Dateien, und sucht nach Zeilen, die mit einem Doppelkreuz (#) in der ersten Spalte beginnen. Die Ausgabe des Präprozessors, wiederum ein C++-Programm, wird zur weiteren Bearbeitung an den Compiler übergeben.

Die Programmiersprache C nutzt den gleichen Präprozessor, so dass alles, was wir hier über den C++-Präprozessor sagen, auch für C gilt.

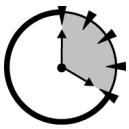
26.2 Die #include-Direktive

Die `#include`-Direktive bindet den Inhalt einer benannten Datei an ihrer Stelle ein. Der Präprozessor versucht nicht, den Inhalt der .h-Datei zu verarbeiten.

Die Include-Datei muss nicht mit .h enden, aber es kann den Programmierer und den Präprozessor verwirren, wenn sie das nicht tut.

Der Name nach dem `#include`-Kommando muss entweder in Hochkommata (" ") oder in Klammern (< >) stehen. Der Präprozessor nimmt an, dass Dateien, die in Hochkommata angegeben werden, benutzerdefiniert sind, und daher im aktuellen Verzeichnis stehen. Nach Dateien in Klammern sucht der Präprozessor in den Verzeichnissen des C++-Compilers.

Die Include-Datei sollte keine C++-Funktionen enthalten, weil sie separat durch das Modul expandiert und kompiliert werden, das die Datei einbindet. Der Inhalt der Include-Datei sollte auf die Klassendefinition, Definition von globalen Variablen und andere Präprozessor-Direktiven beschränkt sein.



20 Min.

26.3 Die Direktive #define

Die Direktive `#define` definiert eine Konstante oder ein Makro. Das folgende Beispiel zeigt, wie `#define` zur Definition einer Konstanten gebraucht wird:

```
#define MAX_NAME_LENGTH 256

void fn(char* pszSourceName)
{
    char szLastName[MAX_NAME_LENGTH];

    if (strlen(pszSourceName) >= MAX_NAME_LENGTH)
    {
        // ... Zeichenkette zu lang -
        // Fehlerbehandlung ...
    }

    // ... hier geht es weiter ...
}
```

Die Präprozessor-Direktive definiert einen Parameter `MAX_NAME_LENGTH`, der zur Compilezeit durch den konstanten Wert 256 ersetzt wird. Der Präprozessor ersetzt den Namen `MAX_NAME_LENGTH` durch die Konstante 256 überall, wo sie benutzt wird. Überall dort, wo wir `MAX_NAME_LENGTH` sehen, sieht der Compiler 256.

Das Beispiel demonstriert die Namenskonvention für `#define`-Konstanten. Namen werden in Großbuchstaben mit Unterstrichen zur Trennung geschrieben.

Wenn sie auf diese Weise verwendet wird, ermöglicht es die `#define`-Direktive dem Programmierer, konstante Werte mit aussagekräftigen Namen zu versehen; `MAX_NAME_LENGTH` sagt dem Programmierer mehr als 256. Konstanten auf diese Weise zu definieren, macht Programme leichter modifizierbar. Z.B. kann die maximale Anzahl Zeichen in einem Namen programmweit limitiert sein. Diesen Wert von 256 auf 128 zu ändern ist einfach, indem nur das `#define`-Kommando geändert werden muss, unabhängig davon, an wie vielen Stellen die Konstante verwendet wird.

26.3.1 Definition von Makros

Die #define-Direktive erlaubt auch Definitionsmakros – eine Compilezeit-Direktive, die Argumente enthält. Das Folgende demonstriert die Definition und die Verwendung des Makros `square( )`, das den Code generiert, um das Quadrat ihres Argumentes zu berechnen.

```
#define square(x) x * x
void fn()
{
    int nSquareOfTwo = square(2);
    // ... usw. ...
}
Der Präprozessor macht hieraus:
void fn()
{
    int nSquareOfTwo = 2 * 2;
    // ... usw. ...
}
```

26.3.2 Häufige Fehler bei der Verwendung von Makros

Der Programmierer muss sehr vorsichtig sein bei der Verwendung von `#define`-Makros. Z.B. erzeugt das Folgende nicht das erwartete Ergebnis:

```
#define square(x) x * x
void fn()
{
    int nSquareOfTwo = square(1 + 1);
}
```

Der Präprozessor generiert hieraus :

```
void fn()
```

```
{
    int nSquareOfTwo = 1 + 1 * 1 + 1;
}
```

Weil Multiplikation Vorrang vor Addition hat, wird der Ausdruck interpretiert, als wenn er so geschrieben wäre:

```
void fn()
{
    int nSquareOfTwo = 1 + (1 * 1) + 1;
}
```

Der Ergebniswert von `nSquareOf` ist 3 und nicht 4.

Eine vollständige Qualifizierung des Makros durch großzügige Verwendung von Klammern schafft Abhilfe, weil Klammern die Reihenfolge der Auswertung kontrollieren. Mit einer Definition von `square` in der folgenden Weise gibt es keine Probleme:

```
#define square(x) ((x) * (x))
```

Doch auch das löst das Problem nicht in jedem Falle. Z.B. kann das Folgende nicht zum Laufen gebracht werden:

```
#define square(x) ((x) * (x))
void fn()
{
    int nV1 = 2;
    int nV2;
    nV2 = square(nV1++);
}
```

Sie können erwarten, dass `nV2` den Ergebniswert 4 statt 6 und `nV1` den Wert 3 statt 4 erhält wegen der folgenden Expansion des Makros:

```
void fn()
{
    int nV1 = 2;
    int nV2;
    nV2 = nV1++ * nV1++;
}
```

Makros sind nicht typsicher. Das kann in Ausdrücken mit unterschiedlichen Typen zu Verwirrungen führen:

```
#define square(x) ((x) * (x))
void fn()
{
    int nSquareOfTwo = square(2.5);
}
```

Weil `nSquareOfTwo` ein `int` ist, könnten Sie erwarten, dass der Ergebniswert 4 statt dem tatsächlichen Wert 6 (2.5 * 2.5 = 6.25) ist.

C++-Inline-Funktionen vermeiden das Problem mit den Makros.

```
inline int square(int x) {return x * x;}
void fn()
{
    int nV1 = square(1 + 1); // nv1 ist 4
    int nV2;
    nV2 = square(nV1++)      // nV2 ist 4, nV1 is 3
    int nV3 = square(2.5)    // nV3 ist 4
}
```

Die Inline-Version von `square( )` erzeugt nicht mehr Code als ein Makro, hat aber nicht die Nachteile der Präprozessor-Variante.

26.4 Compilerkontrolle

Der Präprozessor stellt auch Möglichkeiten bereit, den Compilevorgang zu steuern.

10 Min.

26.4.1 Die #if-Direktive

Die Präprozessor-Direktive, die C++ am ähnlichsten ist, ist die `#if`-Anweisung. Wenn der konstante Ausdruck nach dem `#if` nicht gleich null ist, werden alle Anweisungen bis zu `#else` an den Compiler übergeben. Wenn der Ausdruck null ist, werden die Anweisungen zwischen `#else` und `#endif` übergeben. Der `#else`-Zweig ist optional. Z.B.:

```
#define SOME_VALUE 1
#if SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

wird konvertiert in

```
int n = 1;
```

Ein paar Operatoren sind für den Präprozessor definiert, z.B.

```
#define SOME_VALUE 1
#if SOME_VALUE - 1
int n = 1;
#else
int n = 2;
#endif
wird konvertiert in:
int n = 2;
```

Denken Sie daran, dass dies Entscheidungen zur Compilezeit sind und keine Laufzeitentscheidungen. Die Ausdrücke nach `#if` ***enthalten Konstanten und*** `#define`***-Direktiven – Variablen und Funktionsaufrufe sind nicht erlaubt.***

26.4.2 Die #ifdef-Direktive

Eine andere Kontrolldirektive des Präprozessors ist `#ifdef`. Das `#ifdef` ist wahr, wenn die Konstante danach definiert ist. Somit wird das Folgende

```
#define SOME_VALUE 1
#ifdef SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

konvertiert in:

```
int n = 1;
```

Die Direktive

```
#ifdef SOME_VALUE
int n = 1;
#else
int n = 2;
#endif
```

jedoch wird konvertiert in

```
int n = 2;
```

Das `#ifndef` ist auch definiert mit der genau umgekehrten Definition.

Verwendung von #ifdef/#ifndef zur Einschlusskontrolle

Der häufigste Einsatz von `#ifdef` ist die Kontrolle über die Einbeziehung von Code. Ein Symbol kann nicht mehr als einmal definiert werden. Das Folgende ist ungültig:

```
class MyClass
{
    int n;
};
class MyClass
{
    int n;
};
```

Wenn `MyClass` in der Include-Datei `myclass.h` definiert ist, wäre es ein Fehler, diese Datei zweimal in die .cpp-Quelldatei einzubinden. Sie könnten denken, dass dieses Problem leicht vermeidbar ist. Es ist aber üblich, dass Include-Dateien andere Include-Dateien einbinden, wie in folgendem Beispiel:

```
#include »myclass.h«
class mySpecialClass : public MyClass
{
    int m;
}
```

Ein nichtsahnender Programmierer könnte leicht beide, ass.h und myspecialclass.h, in dieselbe Quelldatei einbinden, was durch die doppelte Definition zu einem Compilerfehler führt.

```
// das kann nicht kompiliert werden
#include »myclass.h«
#include »myspecialclass.h«
void fn(MyClass& mc)
{
    // ... kann ein Objekt aus der Klasse MyClass
    // oder MySpecialClass sein
}
```

Dieses spezielle Beispiel lässt sich leicht korrigieren. In einer großen Anwendung können die Beziehungen zwischen den Include-Dateien viel komplexer sein.

Der wohlüberlegte Einsatz der `#ifdef`-Direktive verhindert dieses Problem, indem myclass.h wie folgt geschrieben wird:

```
#ifndef MYCLASS_H
#define MYCLASS_H
class MyClass
{
    int n;
};
#endif
```

Wenn `myclass.h` zum ersten Mal eingebunden wird, ist `MYCLASS_H` nicht definiert und `#ifndef` ist wahr. Die Konstante `MYCLASS_H` wird jedoch innerhalb von `myclass.h` definiert. Das nächste Mal, wenn `myclass.h` während des Kompilierens angefasst wird, ist `MYCLASS_H` definiert, und die Klassendefinition wird weggelassen.

Debug-Code und #ifdef

Ein anderer häufiger Einsatz von `#ifdef` ist die Integration von Debug-Code zur Compilezeit. Betrachten Sie z.B. die folgende Debug-Funktion:

```
void dumpState(MySpecialClass& msc)
{
    cout <<»MySpecialClass:«
        <<»m = » <<msc.m
        <<»n = » <<msc.n;
}
```

Jedes Mal, wenn diese Funktion aufgerufen wird, druckt `dumpState( )` den Inhalt des `MySpecialClass`-Objektes in die Standardausgabe. Ich kann überall in meinem Programm Aufrufe dieser Funktion einbauen, um den Zustand von `MySpecialClass`-Objekten zu kontrollieren. Wenn das Programm fertig ist, muss ich alle diese Aufrufe wieder entfernen. Das ist nicht nur ermüdend, sondern birgt das Risiko in sich, dass hierbei neue Fehler in das System gelangen. Außerdem kann es sein, dass ich die Anweisungen zum erneuten Debuggen des Systems wieder benötige.

Ich könnte eine Art Flag definieren, das steuert, ob das Programm den Status der `MySpecialClass`-Objekte ausgibt. Aber die Aufrufe selber stellen einen Mehraufwand dar, der die Funktionen verlangsamt. Ein besserer Ansatz ist der folgende:

```
#ifdef DEBUG
void dumpState(MySpecialClass& msc)
{
    cout <<»MySpecialClass:«
        <<»m = » <<msc.m
        <<»n = » <<msc.n;
}
#else
inline dumpState(MySpecialClass& mc)
{
}
#endif
```

0 Min.

Wenn der Parameter `DEBUG` definiert ist, wird die Funktion `dumpState( )` kompiliert. Wenn `DEBUG` nicht definiert ist, wird eine Inline-Version von `dumpState( )` kompiliert, die nichts tut. Der C++-Compiler konvertiert jeden Aufruf dieser Funktion zu nichts.

Wenn die inline-Version der Funktion nicht funktioniert, liegt das vielleicht daran, dass der Compiler keine Inline-Funktionen unterstützt. Dann verwenden Sie die folgende Makrodefinition:
`#define dumpState(x)`
Visual C++ und GNU C++ unterstützen beide diesen Ansatz. Konstanten können in den Projekteinstellungen ohne Hinzufügen der `#define`***-Direktive in den Sourcecode definiert werden. In der Tat ist die Konstante*** `_DEBUG` ***automatisch definiert, wenn im Debug-Modus kompiliert wird.***

Zusammenfassung

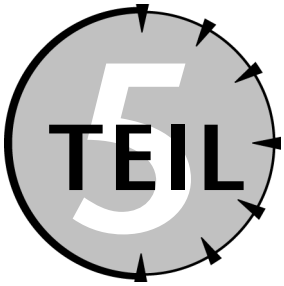
Der häufigste Einsatz des Präprozessors ist das Einbinden der gleichen Klassendefinition oder der gleichen Funktionsprototypen in mehrere .cpp-Quelldateien mit der `#include`-Direktive. Die Präprozessor-Direktiven `#if` und `#ifdef` erlauben die Kontrolle darüber, welche Zeilen des Codes kompiliert werden und welche nicht.

- Der Name der Datei, die mittels `#include` eingebunden wird, sollte mit `.h` enden. Das nicht zu tun, verwirrt andere Programmierer und vielleicht sogar den Compiler. Dateinamen, die in Hochkommata (`""`) eingeschlossen sind, werden im aktuellen (oder in einem anderen benutzerdefinierten) Verzeichnis gesucht, wohingegen Klammern (<>) zur Referenzierung von Include-Dateien von C++ verwendet wird.
- Wenn der konstante Ausdruck nach der `#if`-Direktive nicht null ist, dann werden die folgenden C++-Anweisungen an den Compiler übergeben; andernfalls werden sie nicht übergeben. Die Direktive `#ifdef` x ist wahr, wenn die `#define`-Konstante x definiert ist.
- Alle Präprozessor-Direktiven kontrollieren, welche C++-Anweisungen der Compiler »sieht«. Alle werden zur Compilezeit ausgewertet und nicht zur Laufzeit.

Selbsttest

1. Was ist der Unterschied zwischen `#include "file.h"` und `#include <file.h>`? (Siehe »Die #include-Direktive«)
2. Was sind die beiden Typen der `#define-Direktive`? (Siehe »Die #define-Direktive«)
3. Gegeben die Makrodefinition `#define square(x) x * x`, was ist der Wert von square(2 + 3)? (Siehe »Die #define-Direktive«)
4. Nennen Sie einen Vorteil von Inline-Funktionen gegenüber einer äquivalenten Makrodefinition. (Siehe »Häufige Fehler bei der Verwendung von Makros«)
5. Was ist ein häufiger Einsatz der `#ifndef`-Direktive? (Siehe »Verwendung von #ifdef/#ifndef zur Einschlusskontrolle«)

Sonntagmorgen – Zusammenfassung



1. Wie sieht die Ausgabe des folgenden Programms aus?

```
// ConstructionTest - zeigt die Reihenfolge, in der
//                   Objekte konstruiert werden
#include <stdio.h>
#include <iostream.h>
#include <string.häää

// Advisor - eine leere Klasse
class Advisor
{
  public:
    Advisor(char* pszName)
    {
        cout << »Advisor:«
             << pszName
             << »\n«;
    }
};

class Student
{
  public:
    Student() : adv(»Student Datenelement«)
    {
        cout << »Student\n«;
        new Advisor(»Student lokal«);
    }
    Advisor adv;
};
class GraduateStudent : public Student
{
  public:
    GraduateStudent() :
        adv(»GraduateStudent Datenelement«)
    {
        cout << » GraduateStudent\n«;
        new Advisor(»GraduateStudent lokal«);
    }

  protected:
    Advisor adv;
};

int main(int nArgc, char* pszArgs[])
```

```
{
    GraduateStudent gs;
    return 0;
}
```

2. ***Gegeben sei, dass ein GraduateStudent einen Grad von 2.5 oder besser erreichen muß, um zu bestehen, und 1.5 für reguläre Studenten ausreicht. Schreiben Sie eine Funktion*** `pass( )` ***unter Verwendung der Klassen, die wir in dieser Sitzung geschrieben haben, die einen Grad akzeptiert, und 0 zurückgibt für einen Studenten, der durchgefallen ist, und 1, wenn er bestanden hat.***

3. ***Schreiben Sie eine Klasse*** `Checking` ***(Girokonto), die von*** `Account` ***und*** `CashAccount` ***wie oben gezeigt erbt. Ein Girokono ist einem Sparkonto sehr ähnlich, außer dass eine Gebühr für jedes Abheben anfällt. Machen Sie sich keine Gedanken zu Überziehungen.***

Wenn Sie Zeit sparen möchten, können Sie die Klassen `Account`, `CashAccout` ***und*** `Savings` ***aus dem Verzeichnis ExerciseClasses der beiliegenden CD-ROM kopieren. Sie können diese als Startpunkt verwenden.***

Testen Sie Ihre Klasse mit dem Folgenden:

```
void fn(Account* pAccount)
{
    pAccount->deposit(100);
    pAccount->withdrawal(50);
}

int main(int nArgc, char* pszArgs[])
{
    // eröffne ein Sparkonto
    Savings savings(1234, 0);
    fn(&savings);

    // und nun ein Girokonto
    Checking checking(5678, 0);
    fn(&checking);
// Ausgabe des Ergebnisses
    cout << »Kontostand Sparkonto ist »
         << savings.balance()
         << »\n«;
    cout << »Kontostand Girokonto ist »
         << checking.balance()
         << »\n«;

    return 0;
}
```

Hinweise: Die Ausgabe des Programms sollte so aussehen:

```
Kontostand Sparkonto ist 50
Kontostand Girokonto ist 49
```

4. ***Schreiben Sie ein Programm, das mit Mehrfachvererbung ein Objekt*** `pc` ***aus der Klasse*** `CombinationPrinterCopier` ***erzeugt. Dieses Programm sollte unter Verwendung von*** `pc` ***drucken. Zusätzlich sollte das Programm die elektrische Spannung (voltage) von*** `pc` ***ausgeben.***

   Hinweise:

   a. Suchen Sie zur Hilfe nach Worten in Anführungszeichen.

   b. Sie sind nicht in der Lage, die Spannung durch die Konstruktoren nach oben weiterzugeben. Setzen Sie stattdessen die voltage im Konstruktor `CombinationPrinterCopier`***.***

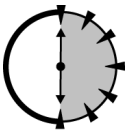
Sonntag-nachmittag

Teil 6

Überladen von Operatoren

Checkliste

- ☑ Überladen von C++-Operatoren im Überblick
- ☑ Diskussion Operatoren und Funktionen
- ☑ Implementierung von Operatoren als Elementfunktion und Nichtelementfunktion
- ☑ Der Rückgabewert eines überladenen Operators
- ☑ Ein Spezialfall: Der Cast-Operator



30 Min.

Die Sitzungen 6 und 7 diskutierten die mathematischen und logischen Operatoren, die C++ für die elementaren Datentypen definiert.

Die *elementaren* Datentypen sind die, die in die Sprache eingebaut sind, wie `int`, `float`, `double` usw. und die Zeigertypen.

Zusätzlich zu den elementaren Operatoren erlaubt es C++ dem Programmierer, Operatoren für Klassen zu definieren, die der Programmierer geschrieben hat. Das wird *Überladen* von *Operatoren* genannt.

Normalerweise ist das Überladen von Operatoren optional und sollte nicht von C++-Anfängern probiert werden. Eine Menge erfahrener C++-Programmierer denken, dass das Überladen von Operatoren keine so tolle Sache ist. Es gibt jedoch drei Operatoren, deren Überladen Sie erlernen müssen: Zuweisung (=), Linksshift (<<) und Rechtsshift (>>). Sie sind wichtig genug, um ein eigenes Kapitel zu bekommen, das diesem Kapitel unmittelbar folgt.

Das Überladen von Operatoren kann Fehler verursachen, die schwer zu finden sind. Seien Sie ganz sicher, wie das Überladen von Operatoren funktioniert, bevor Sie es einsetzen.

27.1 Warum sollte ich Operatoren überladen?

C++ betrachtet benutzerdefinierte Typen, wie `Student` und `Sofa`, als genauso gültig wie die elementaren Typen, z.B. `int` und `char`. Wenn Operatoren für elementare Datentypen definiert sind, warum sollten sie nicht auch für benutzerdefinierte Typen definiert werden können?

Das ist ein schwaches Argument, aber ich gebe zu, dass das Überladen von Operatoren seinen Nutzen hat. Betrachten Sie die Klasse `USDollar`. Einige Operatoren machen keinen Sinn, wenn sie auf `Dollars` angewendet werden. Was soll z.B. das Invertieren eines `Dollars` bedeuten? Auf der anderen Seite sind einige Operatoren definitiv anwendbar. So macht es z.B. Sinn, einen `USDollar` zu einem `USDollar` zu addieren oder davon zu subtrahieren. Es macht Sinn, `USDollar` mit `double` zu multiplizieren. Es macht jedoch keinen Sinn, `USDollar` mit `USDollar` zu multiplizieren.

Das Überladen von Operatoren kann die Lesbarkeit verbessern. Betrachten sie das Folgende, zunächst ohne überladene Operatoren:

```
//expense - berechne bezahlten Betrag
//         (Hauptsumme und Einzelrate)
USDollar expense(USDollar principal, double rate)
{
    // berechne den Ratenbetrag
    USDollar interest = principal.interest(rate);

    // addiere zur Hauptsumme und gib sie zurück
    return principal.add(interest);
}
```

Wenn der entsprechende Operator überladen wird, sieht die gleiche Funktion wie folgt aus:

```
//expense - berechne bezahlten Betrag
//         (Hauptsumme und Einzelrate)
USDollar expense(USDollar principal, double rate)
{
    USDollar interest = principal * rate;
    return principal + interest;
}
```

Bevor wir untersuchen, wie Operatoren überladen werden, müssen wir die Beziehung zwischen Operatoren und Funktionen verstehen.

27.2 Was ist die Beziehung zwischen Operatoren und Funktionen?

Ein Operator ist nichts anderes, als eine eingebaute Funktion mit einer eigenen Syntax. Z.B. hätte der Operator + genauso gut als `add( )` geschrieben werden können.

C++ gibt jedem Operator einen funktionsähnlichen Namen. Der funktionale Name des Operators ist das Operatorsymbol, vor dem das Schlüsselwort `operator` steht, gefolgt von den entsprechenden Argumenttypen. Der Operator + z.B., der ein `int` zu einem anderen `int` addiert und daraus ein `int` erzeugt, heißt `int operator+(int, int)`.

Der Programmierer kann alle Operatoren überladen, außer `.`, `::`, `*` (Dereferenzierung) und `&`, durch Überladen ihres funktionalen Namens mit den folgenden Einschränkungen:

- Der Programmierer kann keine neuen Operatoren einführen. Sie können keinen Operator `x $ y` einführen.
- Das Format der Operatoren kann nicht geändert werden. Somit können Sie keinen Operator `%i` definierten, weil `%` ein binärer Operator ist.
- Der Vorrang der Operatoren kann nicht geändert werden. Ein Programm kann nicht erzwingen, dass `operator+` vor dem `operator*` ausgeführt wird.
- Schließlich können Operatoren nicht neu definiert werden, wenn sie auf elementare Typen angewendet werden. Existierende Operatoren können nur für neue Typen überladen werden.

27.3 Wie funktioniert das Überladen von Operatoren?

Lassen Sie uns das Überladen von Operatoren in Aktion sehen. Listing 27-1 zeigt eine Klasse `USDollar`, die einen Additionsoperator und einen Inkrementoperator definiert.

Listing 27-1: USDollar mit überladenen Operatoren für Addition und Inkrementierung

```
// USDollarAdd - demonstriert die Definition und die
//               Verwendung des Additionsoperators
//               für die Klasse USDollar
#include <stdio.h>
#include <iostream.h>

// USDollar - repräsentiert den US Dollar
class USDollar
{
    // stelle sicher, dass die benutzerdefinierten
    // Operatoren Zugriff auf die protected-Elemente
    // der Klasse haben
    friend USDollar operator+(USDollar&, USDollar&);
    friend USDollar& operator++(USDollar&);

  public:
    // konstruiere ein Dollarobjekt mit initialen
    // Werten für Dollar und Cents
    USDollar(int d = 0, int c = 0);

    // rationalize - normalisiere den Centbetrag
    //               durch Addition eines Dollars pro
    //               100 Cents
    void rationalize()
    {
        dollars += (cents / 100);
        cents   %= 100;
    }

    // output - schreibe den Wert des Objektes
    //          in die Standardausgabe
    void output()
```

```
    {
        cout << »$«
             << dollars
             << ».«
             << cents;
    }

  protected:
    int dollars;
    int cents;
};

USDollar::USDollar(int d, int c)
{
    // speichere die initialen Werte
    dollars = d;
    cents = c;

    rationalize();
}

//operator+ - addiere s1 zu s2 und gib das Ergebnis
//           in einem neuen Objekt zurück
USDollar operator+(USDollar& s1, USDollar& s2)
{
   int cents   = s1.cents   + s2.cents;
   int dollars = s1.dollars + s2.dollars;
   return USDollar(dollars, cents);
}

//operator++ - inkrementiere das Argument; ändere
//            den Wert dieses Objektes
USDollar& operator++(USDollar& s)
{
    s.cents++;
    s.rationalize();
    return s;
}

int main(int nArgc, char* pszArgs[])
{
    USDollar d1(1, 60);
    USDollar d2(2, 50);
    USDollar d3(0, 0);

    // zuerst ein binärer Operator
    d3 = d1 + d2;
    d1.output();
    cout << » + »;
    d2.output();
    cout << » = »;
    d3.output();
    cout << »\n«;

    // jetzt ein unärer Operator
```

```
    ++d3;
    cout << »Nach Inkrementierung gleich »;
    d3.output();
    cout << »\n«;
    return 0;
}
```

Die Klasse `USDollar` ist so definiert, dass sie einen ganzzahligen Dollarbetrag und einen ganzzahligen Centbetrag kleiner 100 speichert. Beim Durcharbeiten der Klasse von vorne nach hinten, sehen wir die Operatoren `operator+( )` und `operator++( )`, die als Freunde der Klasse deklariert sind.

Erinnern Sie sich daran, dass ein Klassenfreund eine Funktion ist, die Zugriff auf die protected-Elemente der Klasse hat. Weil `operator+( )` ***und*** `operator++( )` ***als herkömmliche Nichtelementfunktionen implementiert sind, müssen sie als Freunde der Klasse deklariert sein, um Zugriff auf die protected-Elemente der Klasse zu erhalten.***

Der Konstruktor von `USDollar` erzeugt ein Objekt aus den ganzzahligen Angaben von Dollars und Cents, für die es beide Defaultwerte gibt. Einmal gespeichert ruft der Konstruktor die Funktion `rationalize( )` auf, die den Betrag normalisiert, indem Cent-Anzahlen größer als 100 dem Dollarbetrag zugeschlagen werden. Die Funktion `output( )` schreibt das `USDollar`-Objekt nach `cout`.

Der `operator+( )` wurde mit zwei Argumenten definiert, weil Addition ein binärer Operator ist (d.h. der zwei Argumente hat). Der `operator+( )` beginnt damit, die Beträge von Dollar und Cent ihrer beiden `USDollar`-Argumente zu addieren. Sie erzeugt dann ein neues `USDollar`-Objekt mit diesen Werten und gibt es an den Aufrufenden zurück.

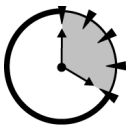
Jede Operation auf einem Wert eines USDollar-Objekts sollte `rationalize( )` ***aufrufen, um sicherzustellen, dass der Centbetrag nicht größer oder gleich 100 ist. Die Funktion*** `operator+( )` ***ruft*** `rationalize( )` ***über den Konstruktor*** `USDollar` ***auf.***

Der Inkrementoperator `operator++( )` hat nur ein Argument. Diese Funktion inkrementiert die Anzahl Cents im Objekt s um eins. Die Funktion gibt dann eine Referenz auf das Objekt zurück, das gerade inkrementiert wurde.

Die Funktion `main( )` zeigt die Summe der beiden Dollarbeträge an. Sie inkrementiert dann das Ergebnis der Addition, und zeigt dieses an. Die Ausführung von `USDollarAdd` sieht wie folgt aus:

```
$1.60 + $2.50 = $4.10
Nach Inkrementierung gleich $4.11
```

Im Gebrauch sehen die Operatoren sehr natürlich aus. Was könnte einfacher sein, als `d3 = d1 + d2` oder `++d3`?



20 Min.

27.3.1 Spezielle Überlegungen

Es gibt ein paar spezielle Überlegungen, die Sie beim Überladen von Operatoren anstellen sollten. Erstens folgert C++ nichts für die Beziehung zwischen Operatoren. Somit hat der Operator `operator+=( )` nichts zu tun mit `operator+( )` oder `operator=( )`.

Zusätzlich führt `operator+(USDollar&, USDollar&)` nicht zwingend eine Addition durch. Sie könnten `operator+( )` auch etwas anderes ausführen lassen; es ist aber eine WIRLICH SCHLECHTE IDEE, das zu tun. Die Leute sind daran gewöhnt, wie sich die Operatoren verhalten. Sie wollen nicht, dass die Operatoren andere Dinge tun.

Ursprünglich war es nicht möglich, den Präfixoperator `++x` unabhängig von Postfixoperator `x++` zu überladen. Genügend Programmierer haben sich darüber beklagt, so dass die Regel aufgestellt wurde, dass `operator++(ClassName)` den Präfixoperator meint und `operator++(ClassName, int)` den Postfixoperator meint; das zweite Argument wird immer mit null belegt. Die gleiche Regel gilt für `operator–( )`.

Wenn Sie nur einen der beiden für `operator++( )` oder `operator–( )` angeben, wird er für beide, Präfix und Postfix, verwendet. Der C++-Standard besagt, dass ein Compiler das nicht tun muss, aber die meisten Compiler tun es.

27.4 Ein detaillierterer Blick

Warum erfolgt bei `operator+( )` eine Wertrückgabe, aber bei `operator++( )` kommt eine Referenz zurück? Das ist kein Zufall, sondern ein sehr wichtiger Unterschied.

Die Addition zweier Objekte verändert keines der Objekte. D.h. `a + b` verändert weder `a` noch `b`. Der `operator+( )` muss daher ein temporäres Objekt erzeugen, in dem er das Ergebnis der Addition speichern kann. Das ist der Grund, weshalb `operator+( )` ein Objekt konstruiert und dieses Objekt als Wert zurückgibt.

Insbesondere würde das Folgende nicht funktionieren:

```
// das funktioniert nicht
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    s1.cents   += s2.cents;
    s1.dollars += s2.dollars;
    return s1;
}
```

weil hierbei `s1` verändert wird. Nach einer Addition `s1 + s2` wäre der Wert von `s1` verändert. Das Folgende funktioniert auch nicht:

```
// das funktioniert auch nicht
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    int cents = s1.cents + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    USDollar result(dollars, cents);
    return result;
}
```

Obwohl das ohne Probleme kompiliert werden kann, erzeugt es ein falsches Ergebnis. Das Problem ist, dass die zurückgegebene Referenz `result` auf ein Objekt verweist, deren Gültigkeitsbereich lokal in der Funktion ist. Somit hat `result` seinen Gültigkeitsbereich bereits verlassen, wenn es von der aufrufenden Funktion verwendet werden kann.

Warum dann nicht einfach einen Speicherbereich vom Heap allozieren?

```
// das funktioniert
USDollar& operator+(USDollar& s1, USDollar& s2)
{
    int cents = s1.cents + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    return *new USDollar(dollars, cents);
}
```

Das wäre gut, außer, dass es keinen Mechanismus gibt, den allozierten Speicher wieder an den Heap zurückzugeben. Solche Speicherlöcher sind schwer zu finden. Ganz langsam geht bei jeder Addition dem Heap ein wenig Speicher verloren.

Die Wertrückgabe zwingt den Compiler, selber ein eigenes temporäres Objekt anzulegen, und es auf den Stack des Aufrufenden zu packen. Das Objekt, das in der Funktion erzeugt wurde, wird dann in das Objekt kopiert, als Teil von `operator+( )`. Aber wie lange existiert das temporäre Objekt von `operator+( )`? Ein temporäres Objekt muss so lange gültig bleiben, bis der »erweiterte Ausdruck«, in dem es vorkommt, fertig ist. Der erweiterte Ausdruck ist alles bis zum Semikolon.

Betrachten Sie z.B. den folgenden Schnipsel:

```
SomeClass f();
LotsAClass g();
void fn()
{
    int i;
    i = f() + (2 * g());

    // ... die temporären Objekte, die f() und g()
    // zurückgeben, sind hier bereits ungültig ...
}
```

Das temporäre Objekt, das von `f( )` zurückgegeben wird, existiert weiter, während `g( )` aufgerufen wird und die Multiplikation durchgeführt wird. Dieses Objekt verliert beim Semikolon seine Gültigkeit.

Um zu unserem `USDollar`-Beispiel zurückzukehren, in dem das temporäre Objekt nicht gesichert wird, funktioniert das Folgende nicht:

```
d1 = d2 + d3 + ++d4;
```

Das temporäre Ergebnis aus der Addition von `d2` und `d3` muss gültig bleiben, während `d4` inkrementiert wird und umgekehrt.

C++ spezifiziert nicht die Reihenfolge, in der Operatoren ausgeführt werden. Somit wissen wir nicht, ob d2 + d3 oder ++d4 zuerst ausgeführt wird. Sie müssen Ihre Funktionen so schreiben, dass es darauf nicht ankommt.

Anders als `operator+( )` modifiziert `operator++( )` sein Argument. Es gibt daher keinen Grund, ein temporäres Objekt zu erzeugen und als Wert zurückzugeben. Das übergebene Argument wird als Referenz an den Aufrufenden zurückgegeben. Die folgende Funktion, die als Wert zurückgibt, enthält einen subtilen Bug:

```
// das ist nicht zu 100% verlässlich
USDollar operator++(USDollar& s)
{
   s.cents++;
   s.rationalize();
   return s;
}
```

Indem `s` als Wert zurückgegeben wird, zwingt die Funktion den Compiler, eine Kopie des Objekts zu machen. In den meisten Fällen ist das in Ordnung. Aber was passiert in einem zugegebenermaßen ungewöhnlichen aber zulässigen Ausdruck wie `++(++a)`? Wir würden erwarten, dass `a` um `2` erhöht wird. Mit der vorangegangenen Definition wird `a` jedoch um `1` erhöht und dann wird eine Kopie von `a` – und nicht `a` selber – um `1` erhöht.

Die allgemeine Regel sieht so aus: Wenn der Operator den Wert des Arguments ändert, übergeben Sie das Argument als Referenz, so dass das Original modifiziert werden und das Argument als Referenz zurückgegeben werden kann, für den Fall, dass das gleiche Objekt in nachfolgenden Operationen verwendet wird. Wenn der Operator nicht den Wert seiner Argumentes verändert, erzeugen Sie ein neues Objekt zur Speicherung des Ergebnisses und geben Sie dieses Objekt als Wert zurück. Die Eingabeargumente können bei Operatoren mit zwei Argumenten immer als Referenzen übergeben werden, um Zeit zu sparen, aber keines der Argumente sollte dann verändert werden.

Es gibt binäre Operatoren, die den Wert ihrer Argumente verändern, wie die speziellen Operatoren +=, *=, usw.

27.5 Operatoren als Elementfunktionen

Ein Operator kann zusätzlich zu seiner Implementierung als Nichtelement eine Elementfunktion sein. Auf diese Art implementiert sieht unser Beispiel `USDollar` wie in Listing 27-2 aus. (Nur die betreffenden Stellen werden gezeigt.)

Die vollständige Version des Programms in Listing 27-2 finden Sie in der Datei USDollarMemberAdd auf der beiliegenden CD-ROM.

Listing 27-2: Implementierung eines Operators als Elementfunktion

```
// USDollar - repräsentiert den US Dollar
class USDollar
{
  public:
    // konstruiere ein Dollarobjekt mit initialen
    // Werten für Dollar und Cents
    USDollar(int d = 0, int c = 0) {
        dollars = d;
        cents = c;

        rationalize();
    }

    // rationalize - normalisiere den Centbetrag
    //               durch Addition eines Dollars pro
    //               100 Cents
    void rationalize()
    {
        dollars += (cents / 100);
        cents   %= 100;
    }

    // output - schreibe den Wert des Objektes
    //          in die Standardausgabe
    void output()
    {
        cout << »$«
             << dollars
             << ».«
             << cents;
    }

    //operator+ - addiere das aktuelle Objekt
    //            zu s2 und gib das Ergebnis
    //            in einem neuen Objekt zurück
    USDollar operator+(USDollar& s2)
    {
        int cents   = this->cents   + s2.cents;
        int dollars = this->dollars + s2.dollars;
        return USDollar(dollars, cents);
    }

    //operator++ - inkrementiere das aktuelle
    //             Objekt
    USDollar& operator++()
    {
        cents++;
        rationalize();
        return *this;
    }

  protected:
    int dollars;
    int cents;
};
```

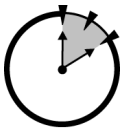
Das Nichtelement `operator+(USDollar, USDollar)` wurde zur Elementfunktion `USDollar::operator+(USDollar)` umgeschrieben. Auf den ersten Blick scheint diese Elementversion ein Argument weniger zu haben als die Nichtelementversion. Wenn Sie zurückdenken, werden Sie sich jedoch daran erinnern, dass `this` das erste, versteckte Argument aller Elementfunktionen ist.

Dieser Unterschied wird am klarsten bei `USDollar::operator+( )` selber. Hier sehen Sie die Nichtelementversion und die Elementversion hintereinander.

```
// operator+ - Nichtelementversion
USDollar operator+(USDollar& s1, USDollar& s2)
{
    int cents   = s1.cents   + s2.cents;
    int dollars = s1.dollars + s2.dollars;
    USDollar t(dollars, cents);
    return t;
}
//operator+ - Elementversion
USDollar USDollar::operator+(USDollar& s2)
{
    int cents   = this->cents   + s2.cents;
    int dollars = this->dollars + s2.dollars;
    USDollar t(dollars, cents);
    return t;
}
```

Wir können sehen, dass die Funktionen fast identisch sind. Jedoch dort, wo die Nichtelementversion `s1` und `s2` addiert, addiert die Elementversion das aktuelle Objekt – auf das `this` zeigt – und `s2`.

Die Elementversion eines Operators hat immer ein Argument weniger als die Nichtelementversion – das Argument auf der linken Seite ist implizit.



10 Min.

27.6 Eine weitere Irritation durch Überladen

Nur weil Sie `operator*(double, USDollar&)` überladen haben, heißt das nicht, dass Sie `operator*(USDollar&, double)` überladen haben. Weil diese Operatoren verschiedene Argumente haben, müssen sie separat überschrieben werden. Das muss nicht so aufwendig sein, wie es vielleicht zuerst aussieht.

Erstens, kann ein Operator sich auf einen anderen Operator beziehen. Im Falle von `operator+( )`, würden wir wahrscheinlich etwas in der folgenden Art tun:

```
USDollar operator*(USDollar& s, double f)
{
    // ... Implementierung der Funktion ...
}
inline USDollar operator*(double f, USDollar& s)
{
    // verwende die obige Definition
    return s * f;
}
```

Die zweite Version ruft einfach die erste Version auf mit der entsprechenden Reihenfolge der Operanden. Die Deklaration als inline spart jeden Zusatzaufwand.

27.7 Wann sollte ein Operator ein Element sein?

Es gibt keinen großen Unterschied zwischen den Implementierungen eines Operators als Element und als Nichtelement mit diesen Ausnahmen:

1. Die folgenden Operatoren müssen als Elementfunktionen implementiert werden:
   = Zuweisung
   ( ) Funktionsaufruf
   [] Indizierung
   -> Klassenzugehörigkeit
2. Ein Operator wie der Folgende kann nicht als Elementfunktion implementiert werden:

```
// operator*(double, USDollar&) - definiert auf
//       Basis von operator*(USDollar&, double)
USDollar operator*(double factor, USDollar& s)
{
    return s * factor;
}
```

Um eine Elementfunktion sein zu können, muss `operator(float, USDollar&)` ein Element der Klasse `double` sein. Wie bereits früher erwähnt, können wir keine Operatoren zu elementaren Klassen hinzufügen. Somit muss ein Operator, für den nur das zweite Argument aus der Klasse ist, als Nichtelement implementiert werden.
Operatoren, die das Objekt, auf dem sie arbeiten, verändern, wie z.B. `operator++( )`, sollten Element der Klasse sein.

27.8 Cast-Operator

Auch der Cast-Operator kann überschrieben werden. Das Programm `USDollarCast` in Listing 27-3 zeigt die Definition und den Gebrauch eines Cast-Operators, der ein `USDollar`-Objekt in ein `double` konvertiert, und wieder zurück.

Listing 27-3: Überladen des Cast-Operators

```
// USDollarCast - demonstriert das Schreiben eines
//                Cast-Operators; dieser konvertiert
//                USDollar nach double, der
//                Konstruktor konvertiert zurück
#include <stdio.h>
#include <iostream.h>

class USDollar
{
  public:
    // Konstruktor, der USDollar aus double erzeugt
    USDollar(double value = 0.0);

    // Cast-Operator
    operator double()
    {
        return dollars + cents / 100.0;
    }
```

```
// display - einfache Debug-Elementfunktion
    void display(char* pszExp, double dV)
    {
        cout << pszExp
             << » = $« << dollars << ».« << cents
             << » (» << dV << »)\n«;
    }

  protected:
    int dollars;
    int cents;
};

// Konstruktor - splitte den double-Wert in
//               ganzzahligen und gebrochenen Teil
USDollar::USDollar(double value)
{
    dollars = (int)value;
    cents = (int)((value - dollars) * 100 + 0.5);
}

int main()
{
    USDollar d1(2.0), d2(1.5), d3, d4;

    // rufe Cast-Operator explizit auf ...
    double dVal1 = (double)d1;
    d1.display(»d1«, dVal1);

    double dVal2 = (double)d2;
    d2.display(»d2«, dVal2);

    d3 = USDollar((double)d1 + (double)d2);
    double dVal3 = (double)d3;
    d3.display(»d3 (Summe d1+d2 mit Casts)«, dVal3);

    //... oder implizit
    d4 = d1 + d2;
    double dVal4 = (double)d3;
    d4.display(»d4 (Summe d1+d2 ohne Casts)«, dVal4);

    return 0;
{
```

Ein Cast-Operator ist das Schlüsselwort `operator`, gefolgt von dem entsprechenden Typ. Die Elementfunktion `USDollar::operator double( )` stellt einen Mechanismus bereit, um ein Objekt der Klasse `USDollar` in ein `double` zu konvertieren. (Aus einem mir unbekannten Grund, haben Cast-Operatoren keinen Rückgabetyp.) Der Konstruktor `USDollar(double)` stellt den Konvertierungspfad von `double` zu `USDollar` her.

Wie das vorangegangene Beispiel zeigt, kann die Konvertierung mittels Cast-Operators entweder explizit oder implizit aufgerufen werden. Lassen Sie uns den impliziten Fall genauer betrachten.

Um den Ausdruck `d4 = d1 + d2` im Programm `USDollarCast` mit Sinn zu versehen, durchläuft C++ die folgenden Schritte:

1. Zuerst sucht C++ nach einer Elementfunktion `USDoller::operator+(USDollar)`.
2. Wenn diese nicht gefunden werden konnte, sucht C++ nach der Nichtelementversion der gleichen Sache, d.h. `operator+(USDoller, USDollar)`.
3. Weil auch diese Version fehlt, sucht C++ nach einem `operator+( )`, den es unter Konvertierung des einen oder anderen Arguments in einen anderen Typ verwenden könnte. Schließlich findet es etwas Passendes: Wenn beide, `d1` und `d2`, nach `double` konvertiert werden, kann der eingebaute `operator+(double, double)` verwendet werden. Selbstverständlich muss das Ergebnis mittels des Konstruktors von `double` nach `USDollar` konvertiert werden.

Die Ausgabe von `USDollarCast` finden Sie unten. Die Funktion `USDollar::cast( )` erlaubt es dem Programmierer, frei zwischen `USDollar`-Objekten und `double`-Werten hin und her zu konvertieren.

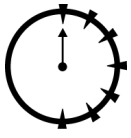
```
d1 = $2.0 (2)
d2 = $1.50 (1.5)
d3 (Summe d1+d2 mit Casts) = $3.50 (3.5)
d4 (Summe d1+d2 ohne Casts) = $3.50 (3.5)
```

Das zeigt sowohl den Vorteil als auch den Nachteil davon, Cast-Operatoren bereitzustellen. Die Bereitstellung eines Konvertierungspfades von `USDollar` nach `double` befreit den Programmierer davon, einen vollständigen Satz von Operatoren bereitstellen zu müssen. `USDollar` kann einfach auf die für `double` definierten Operatoren zurückgreifen.

Auf der anderen Seite nimmt das dem Programmierer die Kontrolle darüber, welche Operatoren definiert werden. Durch den Konvertierungspfad nach `double`, bekommt `USDollar` alle Operatoren von `double`, ob sie nun Sinn machen oder nicht. Ich hätte genauso gut `d4 = d1 * d2` schreiben können. Außerdem kann es sein, dass diese zusätzliche Konvertierung nicht besonders schnell ist. Diese einfache Addition z.B. enthält drei Typkonvertierungen mit all den verbundenen Funktionsaufrufen, Multiplikationen, Divisionen usw.

Passen Sie auf, dass Sie nicht zwei Konvertierungspfade zum gleichen Typ bereitstellen. Das Folgende muss Probleme erzeugen:

```
class A
{
  public:
    A(B& b);
};
class B
{
  public:
    operator A();
};
```



0 Min.

Wenn ein Objekt der Klasse `B` in ein Objekt der Klasse `A` konvertiert werden soll, weiß der Compiler nicht, ob er den Cast-Operator `B::operator A( )` von `B` oder den Konstruktor `a::A(B&)` von `A` verwenden soll, die beide von einem `B` ausgehen und bei einem `A` ankommen.

Vielleicht ist das Ergebnis der beiden Pfade das Gleiche, aber der Compiler weiß das nicht. C++ muss wissen, welchen Konvertierungspfad Sie meinen. Der Compiler gibt eine Meldung aus, wenn er den Pfad nicht unzweideutig bestimmen kann.

Zusammenfassung

Eine neue Klasse mit den entsprechenden Operatoren zu überladen, kann zu einfachem und elegantem Anwendungscode führen. In den meisten Fällen ist das Überladen von Operatoren jedoch nicht nötig. Die folgenden Sitzungen untersuchen zwei Fälle, in denen das Überladen von Operatoren kritisch ist.

- Das Überladen von Operatoren ermöglicht es dem Programmierer, existierende Operatoren für seine eigenen Klassen neu zu definieren. Der Programmierer kann jedoch keine neuen Operatoren hinzufügen, noch die Syntax bestehender Operatoren ändern.
- Es ist ein entscheidender Unterschied zwischen Übergabe und Rückgabe eines Objekts als Wert oder als Referenz. Abhängig vom Operator kann dieser Unterschied kritisch sein.
- Operatoren, die das Objekt verändern, sollten als Element implementiert werden. Einige Operatoren müssen als Element implementiert werden. Operatoren, bei denen auf der linken Seite ein elementarer Datentyp und keine benutzerdefinierte Klasse steht, können nicht als Elementfunktionen implementiert werden. Ansonsten macht es keinen großen Unterschied.
- Der Cast-Operator erlaubt es dem Programmierer, C++ mitzuteilen, wie ein benutzerdefiniertes Klassenobjekt in einen elementaren Typ konvertiert werden kann. Z.B. könnte die Konvertierung von `Student` nach `int` die ID des Studenten zurückgeben (ich habe nicht gesagt, dass diese Konvertierung eine gute Idee ist, sondern nur, dass sie möglich ist.) Dann können eigene Klassen mit den elementaren Datentypen in Ausdrücken gemischt werden.
- Benutzerdefinierte Operatoren erlauben es dem Programmierer, Programme zu schreiben, die leichter zu lesen und zu pflegen sind. Eigene Operatoren können jedoch trickreich sein und sollten mit Vorsicht verwendet werden.

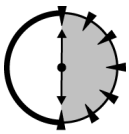
Selbsttest

1. Es ist wichtig, dass Sie drei Operatoren für jede Klasse überschreiben können. Welche Operatoren sind das? (Siehe Einleitung)
2. Wie könnte der folgende Code Sinn machen? (Siehe »Warum soll ich Operatoren überladen?«)
   ```
   USDollar dollar(100, 0);
   DM& mark = !dollar;
   ```
3. Gibt es einen anderen Weg, das Obige nur durch »normale« Funktionsaufrufe zu schreiben, ohne Operatoren zu verwenden, die vom Programmierer geschrieben wurden? (Siehe »Warum soll ich Operatoren überladen?«)

Der Zuweisungs-operator

Checkliste

- ☑ Einführung in den Zuweisungsoperator
- ☑ Warum und wann der Zuweisungsoperator nötig ist
- ☑ Ähnlichkeiten von Zuweisungsoperator und Kopierkonstruktor



30 Min.

Ob Sie nun anfangen, Operatoren zu überladen oder nicht, Sie müssen schon früh lernen, den Zuweisungsoperator zu überladen. Der Zuweisungsoperator kann für jede benutzerdefinierte Klasse überladen werden. Wenn Sie sich an das hier vorgestellte Muster halten, werden Sie sehr bald Ihre eigene Version von `operator=( )` schreiben.

28.1 Warum ist das Überladen des Zuweisungsoperators kritisch?

C++ stellt eine Defaultdefinition von `operator=( )` für alle benutzerdefinierten Klassen bereit. Diese Defaultdefinition erstellt eine Element-zu-Element-Kopie, so ähnlich wie der Kopierkonstruktor. In folgendem Beispiel werden alle Elemente von `source` über die entsprechenden Elemente von `destination` kopiert.

```
void fn()
{
    MyStruct source, destination;
    destination = source;
}
```

Diese Defaultimplementierung ist jedoch nicht korrekt für Klassen, die Ressourcen allozieren wie z.B. Speicher vom Heap. Der Programmierer muss `operator=( )` überladen, um den Transfer von Ressourcen zu realisieren.

28.1.1 Vergleich mit Kopierkonstruktor

Der Zuweisungsoperator ist dem Kopierkonstruktor sehr ähnlich. Eingesetzt sehen die beiden fast identisch aus.

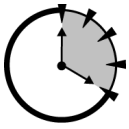
```
void fn(MyClass &mc)
{
    MyClass newMC(mc);    // klar, das verwendet den
                          // Kopierkonstruktor
    MyClass newerMC = mc; // weniger klar, das ruft
                          // auch Kopierkonstruktor
    MyClass newestMC;     // das erzeugt
                          // Default-Objekt
    newestMC = mc;        // und überschreibt es mit
                          // dem Argument
}
```

Die Erzeugung von `newMC` folgt dem Standardmuster, ein neues Objekt unter Verwendung des Kopierkonstuktors `MyClass(MyClass&)` als ein Spiegelbild des Originals zu erzeugen. Nicht so offensichtlich ist, dass C++ das zweite Format erlaubt, bei dem `newerMC` mittels Kopierkonstruktor erzeugt wird.

`newestMC` wird mittels des Default-Konstruktors erzeugt und dann durch den Zuweisungsoperator mit `mc` überschrieben. Der Unterschied ist, dass bei Aufruf des Kopierkonstruktors für `newerMC` dieses Objekt noch nicht existierte. Bei Aufruf des Zuweisungsoperators für `newestMC` war es bereits ein `MyClass-Objekt` im besten Sinne.

Die Regel sieht so aus: Der Kopierkonstruktor wird benutzt, wenn ein neues Objekt erzeugt wird. Der Zuweisungsoperator wird verwendet, wenn das Objekt auf der linken Seite bereits existiert.

Wie der Kopierkonstruktor sollte ein Zuweisungsoperator immer dann bereitgestellt werden, wenn eine flache Kopie nicht angebracht ist. (Sitzung 20 enthält eine umfangreiche Diskussion von flachen und tiefen Konstruktoren.) Es reicht aus zu sagen, dass ein Kopierkonstruktor und ein Zuweisungsoperator dann definiert werden sollten, wenn die Klasse Ressourcen alloziert, damit es nicht dazu kommt, dass zwei Objekte auf die gleichen Ressourcen zeigen.



20 Min.

28.2 Wie den Zuweisungsoperator überladen?

Den Zuweisungsoperator zu überladen funktioniert so, wie bei den anderen Operatoren. Das Beispielprogramm `DemoAssign`, das Sie in Listing 28-1 finden, enthält sowohl einen Kopierkonstruktor als auch einen Zuweisungsoperator.

Denken Sie daran, dass der Zuweisungsoperator ein Element der Klasse sein muss.

Listing 28-1: Überladen des Zuweisungsoperators

```
// DemoAssign - demonstrate the assignment operator
#include <stdio.h>
#include <string.h>
#include <iostream.h>

// Name - eine generische Klasse, die den
//        Zuweisungsoperator und den
//        Kopierkonstruktor demonstriert
class Name
{
  public:
    Name(char *pszN = 0)
    {
        copyName(pszN);
    }
    Name(Name& s)
    {
        copyName(s.pszName);
    }
    ~Name()
    {
        deleteName();
    }
    // Zuweisungsoperator
    Name& operator=(Name& s)
    {
        // gib Altes frei ...
        deleteName();
        //... bevor es durch Neues ersetzt wird
        copyName(s.pszName);
        // gib Referenz auf Objekt zurück
        return *this;
    }

    // display - gibt das Objekt in die
    //           Standardausgabe aus
    void display()
    {
        cout << pszName;
    }

  protected:
    void copyName(char *pszN);
    void deleteName();
    char *pszName;
};

// copyName() - alloziere Heapspeicher zum Speichern
void Name::copyName(char *pszName)
{
    this->pszName = 0;
    if (pszName)
    {
```

```
        this->pszName = new char[strlen(pszName) + 1];
        strcpy(this->pszName, pszName);
    }
}

// deleteName() - gib Heapspeicher zurück
void Name::deleteName()
{
    if (pszName)
    {
        delete pszName;
        pszName = 0;
    }
}

// displayNames - Ausgabefunktion, um die Zeilen in
//                main() zu reduzieren
void displayNames(Name& pszN1, char* pszMiddle,
                  Name& pszN2, char* pszEnd)
{
    pszN1.display();
    cout << pszMiddle;
    pszN2.display();
    cout << pszEnd;
}

int main(int nArg, char* pszArgs[])
{
    // erzeuge zwei Objekte
    Name n1(»Claudette«);
    Name n2(»Greg«);
    displayNames(n1, » und »,
                 n2, » sind neu erzeugte Objekte\n«);

    // mache eine Kopie eines Objektes
    Name n3(n1);
    displayNames(n3, » ist eine Kopie von »,
                 n1, »\n«);

    // mache eine Kopie des Objektes von der
    // Adresse aus
    Name* pN = &n2;
    Name n4(*pN);
    displayNames(n4, » ist eine Kopie (Adresse) von«,
                 n2, »\n«);

    // überschreibe n2 mit n1
    n2 = n1;
    displayNames(n1, » wurde zugewiesen an »,
                 n2, »\n«);
    return 0;
}
```

Ausgabe:

```
Claudette und Greg sind neu erzeugte Objekte
Claudette ist eine Kopie von Claudette
Greg ist eine Kopie (Adresse) von Greg
Claudette wurde zugewiesen an Claudette
```

Die Klasse `Name` hält den Namen einer Person im Speicher, der vom Heap alloziert wurde. Die Konstruktoren und der Destruktor der Klasse `Name` sind denen sehr ähnlich, die in den Sitzungen 19 und 20 vorgestellt wurden. Der Konstruktor `Name(char*)` kopiert den gegebenen Namen in das Datenelement `pszName`. Dieser Konstruktor ist auch der Defaultkonstruktor. Der Kopierkonstruktor `Name(&Name)` kopiert den Namen des übergebenen Objektes in den Namen des aktuellen Objektes durch einen Aufruf der Funktion `copyName( )`. Der Destruktor gibt die `pszName`-Zeichenkette durch einen Aufruf von `deleteName( )` an den Heap zurück.

Die Funktion `main( )` demonstriert jede dieser Elementfunktionen. Die Ausgabe von `DemoAssign` finden Sie oben am Ende von Listing 28-1.

Schauen Sie sich den Zuweisungsoperator genau an. Die Funktion `operator=( )` sieht doch wirklich aus wie ein Destruktor, unmittelbar gefolgt von einem Kopierkonstruktor. Das ist typisch. Betrachten Sie die Zuweisung im Beispiel `n2 = n1`. Das Objekt `n2` hat bereits einen Namen (»Greg«). In der Zuweisung muss der Speicher, den der ursprüngliche Name belegt, an den Heap zurückgegeben werden durch einen Aufruf von `deleteName( )`, bevor neuer Speicher mittels `copyName( )` alloziert und zugewiesen wird, in dem der neue Name (»Claudette«) gespeichert wird.

Der Kopierkonstruktor musste `deleteName( )` nicht aufrufen, weil das Objekt noch nicht existierte. Es war daher noch kein Speicher belegt, als der Konstruktor aufgerufen wurde.

Im Allgemeinen hat ein Zuweisungsoperator zwei Teile. Der erste Teil baut den Destruktor in dem Sinne nach, dass er die belegten Ressourcen des Objektes freigibt. Der zweite Teil baut den Kopierkonstruktor nach in dem Sinne, dass er neue Ressourcen alloziert.

28.2.1 Zwei weitere Details des Zuweisungsoperators

Es gibt zwei weitere Details des Zuweisungsoperators, die Sie kennen sollten. Erstens ist der Rückgabetyp von `operator=( )` gleich `Name&`. Ich bin darauf nicht im Detail eingegangen, aber der Zuweisungsoperator ist ein Operator wie jeder andere. Ausdrücke, die einen Zuweisungsoperator enthalten, haben einen Wert und einen Typ, wobei diese beiden vom endgültigen Wert auf der linken Seite stammen. Im folgenden Beispiel ist der Wert von `operator=( )` gleich 2.0 und der Typ ist `double`:

```
double d1, d2;
void fn(double );
d1 = 2.0;
```

Dadurch wird es möglich, dass der Programmierer schreiben kann:

```
d2 = d1 = 2.0
fn(d2 = 3.0); // führt die Zuweisung aus, und
              // übergibt den Ergebniswert an fn()
```

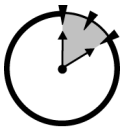
Der Wert 2.0 der Zuweisung `d1 = 2.0` und ihr Typ `double` werden an den nächsten Zuweisungsoperator übergeben. Im zweiten Beispiel wird der Wert der Zuweisung `d2 = 3.0` an die Funktion `fn( )` übergeben.

Ich hätte auch `void` zum Rückgabetyp von `Name::operator=( )` machen können. Wenn ich das jedoch tue, funktioniert das obige Beispiel nicht mehr:

```
void otherFn(Name&);
void fn()
{
    Name n1, n2, n3;

    // das Folgende ist nur möglich, wenn der
    // Zuweisungsoperator eine Referenz auf
    // das aktuelle Objekt zurückgibt
    n1 = n2 = n3;
    otherFn(n1 = n2);
}
```

Das Ergebnis der Zuweisung `n1 = n2` ist `void` – der Rückgabetyp von `operator=( )` – was nicht mit dem Prototyp von `otherFn( )` übereinstimmt. Die Deklaration von `operator=( )` mit einer Referenz auf das aktuelle Objekt und die Rückgabe von `*this` bleiben die Semantik für den Zuweisungsoperator bei elementaren Typen.

Das zweite Detail ist, dass `operator=( )` als Elementfunktion geschrieben wurde. Anders als andere Operatoren kann der Zuweisungsoperator nicht mit einer Nichtelementfunktion überladen werden. Die speziellen Zuweisungsoperatoren, wie `+=` und `*=`, haben keine besonderen Einschränkungen und können Nichtelemente sein.



10 Min.

28.3 Ein Schlupfloch

Ihre Klasse mit einem Zuweisungsoperator auszustatten, kann ihren Anwendungscode sehr flexibel machen. Wenn Ihnen das jedoch zu viel ist, oder wenn Sie keine Kopien Ihrer Objekte machen können, verhindert das Überladen des Zuweisungsoperators mit einer protected-Funktion, dass jemand versehentlich eine flache Kopie eines Objektes erstellt. Z.B.:

```
class Name
{
    //... wie zuvor ...
  protected:
    // Zuweisungsoperator
    Name& operator=(Name& s)
    {
        return *this;
    }
};
```

Mit dieser Definition, werden Zuweisungen wie die folgende verhindert:

```
void fn(Name &n)
{
    Name newN;
    newN = n;   // erzeugt einen Compilerfehler -
                // Funktion hat keinen Zugriff
                // auf operator=()
}
```

0 Min.

Dieser Kopierschutz für Klassen erspart Ihnen den Ärger mit dem Überladen des Zuweisungsoperators, aber reduziert die Flexibilität Ihrer Klasse.

Wenn Ihre Klasse Ressourcen alloziert, wie z.B. Speicher vom Heap, müssen Sie entweder einen entsprechenden Zuweisungsoperator und Kopierkonstruktor schreiben oder beide protected machen und verhindern, dass die von C++ bereitgestellte Defaultmethode verwendet wird.

Zusammenfassung

Der Zuweisungsoperator ist der einzige Operator, den Sie überschreiben müssen, aber nur unter bestimmten Bedingungen. Glücklicherweise ist es nicht schwer, einen Zuweisungsoperator für Ihre Klasse zu definieren, wenn Sie dem Muster folgen, das in dieser Sitzung beschrieben wurde.

- C++ stellt einen Default-Zuweisungsoperator bereit, der eine Element-zu-Element-Kopie durchführt. Diese Version der Zuweisung ist für viele Klassentypen in Ordnung; Klassen jedoch, die Ressourcen allozieren, müssen einen Kopierkonstruktor und einen überladenen Zuweisungsoperator enthalten.
- Die Semantik des Zuweisungsoperators entspricht im Wesentlichen einem Destruktor, gefolgt von einem Kopierkonstruktor. Der Destruktor entfernt alle Ressourcen, die möglicherweise bereits existieren, während der Kopierkonstruktor eine tiefe Kopie der zugewiesenen Ressourcen erstellt.
- Den Zuweisungsoperator protected zu deklarieren, reduziert die Gefahr, aber beschränkt Ihre Klasse, indem mit Ihrer Klasse keine Zuweisungen ausgeführt werden können.

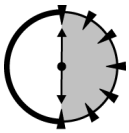
Selbsttest

1. Wann muss Ihre Klasse einen Zuweisungsoperator enthalten? (Siehe »Warum ist das Überladen des Zuweisungsoperators kritisch?«)
2. Der Rückgabetyp des Zuweisungsoperators sollte immer mit dem Klassentyp übereinstimmen. Warum? (Siehe »Zwei weitere Details des Zuweisungsoperators«)
3. Wie können Sie verhindern, einen Zuweisungsoperator schreiben zu müssen? (Siehe »Ein Schlupfloch«)

Stream-I/O

Checkliste

- ☑ Stream-I/O als überladenen Operator wiederentdecken
- ☑ Streamdatei-I/O verwenden
- ☑ Streampuffer-I/O verwenden
- ☑ Eigene Inserter und Extraktor schreiben
- ☑ Hinter den Kulissen von Manipulatoren



30 Min.

Bis jetzt haben alle Programme ihre Eingaben über das `cin`-Eingabeobjekt und ihre Ausgaben über das `cout`-Ausgabeobjekt erledigt. Vielleicht haben Sie nicht viel darüber nachgedacht, aber diese Technik der Eingabe/Ausgabe ist eine Teilmenge dessen, was als Stream-I/O bezeichnet wird.

Diese Sitzung erklärt Stream-I/O im Detail. Ich muss Sie warnen: Stream-I/O ist ein zu großes Thema, um in einer einzigen Sitzung behandelt werden zu können – ganze Bücher sind diesem Thema gewidmet. Ich kann Ihnen jedoch zu einem Anfang verhelfen, so dass Sie die Hauptoperationen durchführen können.

29.1 Wie funktioniert Stream-I/O?

Stream-I/O basiert auf überladenen Versionen von `operator>>( )` und `operator<<( )`. Die Deklaration dieser überladenen Operatoren finden sich in der Include-Datei iostream.h, die wir in unsere Programme seit Sitzung 2 eingebunden haben. Der Code für diese Funktionen ist in der Standardbibliothek von C++ enthalten, mit der Ihr C++-Programm gelinkt wird. Das Folgende zeigt ein paar Prototypen aus iostream.h:

```
// für die Eingabe haben wir:
istream& operator>>(istream& source, char *pDest);
istream& operator>>(istream& source, int  &dest);
istream& operator>>(istream& source, char &dest);
//... usw. ...

// für die Ausgabe haben wir:
```

```
ostream& operator<<(ostream& dest, char *pSource);
ostream& operator<<(ostream& dest, int   source);
ostream& operator<<(ostream& dest, char  source);
//... usw. ...
```

Wenn `operator>>( )` für I/O überladen wird, wird er *Extraktor* genannt; `operator<<( )` wird *Inserter* genannt.

Lassen Sie uns im Detail ansehen, was passiert, wenn ich Folgendes schreibe:

```
#include <iostream.h>
void fn()
{
    cout << »Ich heiße Randy\n«;
}
```

Das Objekt `cout` ist ein Objekt der Klasse `ostream` (mehr dazu später). Somit bestimmt C++, dass die Funktion `operator<<(ostream&, char*)` am besten übereinstimmt. C++ erzeugt einen Aufruf dieser Funktion, dem so genannten `char*`-Inserter und übergibt der Funktion das `ostream`-Objekt `cout` und die Zeichenkette `»Ich heiße Randy\n«` als Argument. D.h. es wird aufgerufen `operator<<(cout, »Ich heiße Randy\n«)`. Der `char*`-Inserter, der Teil der C++-Standardbibliothek ist, führt die angeforderte Ausgabe durch.

Die Klassen `ostream` und `istream` sind die Basis für eine Menge von Klassen, die den Anwendungscode mit der Außenwelt verbinden, Eingabe vom und Ausgabe ins Dateisystem eingeschlossen. Woher wusste der Compiler, dass `cout` aus der Klasse `ostream` ist? Diese und einige andere globale Objekte sind in `iostream.h` deklariert. Eine Liste dieser Objekte finden Sie in Tabelle 29-1. Diese Objekte werden bei Programmstart automatisch erzeugt, bevor `main( )` die Kontrolle erhält.

Tabelle 29-1: Objekte der Standard-Stream-I/O

Objekt	Klasse	Aufgabe
`cin`	`istream`	Standardeingabe
`cout`	`ostream`	Standardausgabe
`cerr`	`ostream`	Standardfehlerausgabe
`clog`	`ostream`	Standarddruckerausgabe

Unterklassen von `ostream` und `istream` werden für die Eingabe von und die Ausgabe in Dateien und interne Puffer verwendet.

29.2 Die Unterklassen fstream

Die Unterklassen `ofstream`, `ifstream` und `fstream` sind in der Include-Datei `fstream.h` definiert, um Streameingabe und Streamausgabe für Dateien zu realisieren. Diese drei Klassen bieten eine Vielzahl von Elementfunktionen. Eine vollständige Liste finden Sie in der Dokumentation Ihres Compilers, aber lassen Sie mich eine kurze Einführung geben.

Die Klasse `ofstream`, die für die Dateiausgabe verwendet wird, hat mehrere Konstruktoren, von denen der folgende der nützlichste ist:

```
ofstream::ofstream(char *pszFileName,
                   int mode = ios::out,
                   int prot = filebuff::openprot);
```

Das erste Argument ist ein Zeiger auf den Namen der Datei, die geöffnet werden soll. Das zweite und dritte Argument geben an, wie die Datei geöffnet werden soll. Die gültigen Werte für `mode` finden Sie in Tabelle 29-2 und die für `prot` in Tabelle 29-3. Diese Werte sind Bitfelder, die durch OR verbunden sind (die Klassen `ios` und `filebuff` sind beide Elternklasse von `ostream`).

Der Ausdruck `ios::out` ***bezieht sich auf ein statisches Element der Klasse*** `ios`.

Tabelle 29-2: Konstanten zur Kontrolle, wie Dateien geöffnet werden

Flag	Bedeutung
`ios::ate`	Anhängen ans Ende der Datei, falls sie existiert
`ios::in`	Datei zur Eingabe öffnen (implizit für istream)
`ios::out`	Datei zur Ausgabe öffnen (implizit für ostream)
`ios::trunc`	Schneide Datei ab, falls sie existiert (default)
`ios::nocreate`	Wenn Datei nicht bereits existiert, gibt Fehler zurück
`ios::noreplace`	Wenn Datei bereits existiert, gibt Fehler zurück
`ios::binary`	Öffne Datei im Binärmodus (Alternative ist Textmodus)

Tabelle 29-3: Werte für prot im Konstruktor ofstream

Flag	Bedeutung
`filebuf::openprot`	Kompatibilitäts-Sharing-Modus
`filebuf::sh_none`	Exklusiv – kein Sharing
`filebuf::sh_read`	Lese-Sharing erlaubt
`filebuf::sh_write`	Schreib-Sharing erlaubt

Das folgende Programm z.B. öffnet eine Datei MYNAME und schreibt ein paar wichtige und absolut der Wahrheit entsprechende Informationen hinein:

```
#include <fstream.h>
void fn()
{
   // öffne die Textdatei MYNAME zum Schreiben -
   // überschreibe, was in der Datei steht
   ofstream myn(»MYNAME«);
   myn << »Randy Davis ist höflich und hübsch\n«;
}
```

Der Konstruktor `ofstream::ofstream(char*)` erwartet nur einen Dateinamen und stellt Default-Werte für die anderen Dateimodi bereit. Wenn die Datei MYNAME bereits existiert, wird sie geleert; anderenfalls wird MYNAME erzeugt. Zusätzlich wird die Datei im Kompatibilitäts-Sharing-Modus geöffnet.

Wenn ich eine Datei im Binärmodus öffnen und an das Ende der Datei anfügen möchte, wenn die Datei bereits existiert, würde ich wie folgt ein `ostream`-Objekt erzeugen (siehe Tabelle 29-2). (Im Binärmodus werden Zeilenumbrüche bei der Ausgabe nicht in Carriage Return und Line Feed verwandelt, und die umgekehrte Konvertierung findet bei der Eingabe ebenfalls nicht statt.)

```
void fn()
{
   // öffne die Binärdatei BINFILE zum Schreiben;
   // wenn sie bereits existiert, füge ans Ende an
   ofstream bfile(»BINFILE«, ios::binary | ios::ate);
   //... Fortsetzung wie eben ...
}
```

Die Streamobjekte enthalten Zustandsinformationen über ihren I/O-Prozess. Die Elementfunktion `bad( )` gibt ein Fehlerflag zurück, das innerhalb der Klassen geführt wird. Das Flag ist nicht null, wenn das Dateiobjekt einen Fehler enthält.

Streamausgabe geht der Ausnahmen-basierten Technik der Fehlerbehandlung voraus, die in Sitzung 30 erklärt wird.

Um zu überprüfen, ob die Dateien MYNAME und BINFILE in dem früheren Beispiel korrekt geöffnet wurden, könnte ich schreiben:

```
#include <fstream.h>
void fn()
{
    ofstream myn(»MYNAME«);
    if (myn.bad()) // wenn das Öffnen fehlschlägt ...
    {
        cerr << »Fehler beim Öffnen von MYNAME\n«;
        return;    //... Fehler ausgeben und fertig
   }
    myn << »Randy Davis ist höflich und hübsch\n«;
}
```

Alle Versuche, Ausgaben mit einem `ofstream`-Objekt durchzuführen, das einen Fehler enthält, haben keinen Effekt, bis der Fehler durch einen Aufruf der Elementfunktion `clear( )` gelöscht wird.

Dieser letzte Paragraph ist wörtlich gemeint – es ist keine Ausgabe möglich, so lange das Fehlerflag nicht null ist.

Der Destruktor der Klasse `ofstream` schließt die Datei automatisch. Im vorangegangenen Beispiel wurde die Datei bei Verlassen der Funktion geschlossen.

Die Klasse `ifstream` arbeitet auf die gleiche Weise bei der Eingabe, wie das folgende Beispiel zeigt:

```
#include <fstream.h>
void fn()
{
   // öffnet Datei zum Lesen; erzeuge die
   // Datei nicht, wenn sie nicht existiert
   ifstreambankStatement(»STATEMNT«, ios::nocreate);
   if (bankStatement.bad())
   {
       cerr << »Datei STATEMNT nicht gefunden\n«;
       return;
   }
   while (!bankStatement.eof())
   {
      bankStatement >> nAccountNumber >> amount;
      // ... verarbeite Abhebung
   }
}
```

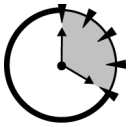
Die Funktion öffnet die Datei STATEMNT durch die Erzeugung des Objektes `bankStatement`. Wenn die Datei nicht existiert, wird sie erzeugt. (Wir nehmen an, dass die Datei Informationen für uns hat, es würde daher keinen Sinn machen, eine neue, leere Datei zu erzeugen.) Wenn das Objekt fehlerhaft ist (z.B. wenn das Objekt nicht erzeugt wurde), gibt die Funktion eine Fehlermeldung aus und beendet die Ausführung. Andernfalls durchläuft die Funktion eine Schleife und liest dabei `nAccount`Number und den Abhebungsbetrag `amount`, bis die Datei leer ist (end-of-file ist wahr).

Der Versuch, aus einem `ifstream`-Objekt zu lesen, das einen Fehler enthält, kehrt sofort zurück, ohne etwas gelesen zu haben.

Lassen Sie mich erneut warnen. Es wird nicht nur nichts zurückgegeben, wenn aus einem Eingabestream gelesen wird, der einen Fehler enthält, sondern der Puffer kommt unverändert zurück. Das Programm kann leicht den falschen Schluss daraus ziehen, dass die gleiche Eingabe wie zuvor gelesen wurde. Schließlich wird `eof( )` ***nie true liefern auf einem Stream, der sich im Fehlerzustand befindet.***

Die Klasse `fstream` ist wie eine Klasse, die `ifstream` und `ofstream` kombiniert (in der Tat erbt sie von beiden). Ein Objekt der Klasse `fstream` kann zur Eingabe oder zur Ausgabe geöffnet werden oder für beides.

29.3 Die Unterklassen strstream



20 Min.

Die Klassen `istrstream`, `ostrstream` und `strstream` sind in der Include-Datei strstrea.h definiert. (Der Dateiname erscheint unter MS-DOS abgeschnitten, weil dort nicht mehr als 8 Zeichen pro Dateiname erlaubt sind; GNU C++ verwendet den vollständigen Namen strstream.h.)

Diese Klassen erlauben die Operationen, die in den `fstream`-Klassen für Dateien definiert sind, für Puffer, die sich im Speicher befinden.

Der Codeschnipsel parst die Daten einer Zeichenkette unter Verwendung von Streameingabe:

```
#include <strstrea.h>
// <strstream.h>  für GNU C++
char* parseString(char *pszString)
{
    // assoziiere ein istrstream-Objekt mit der
    // Eingabezeichenkette
    istrstream inp(pszString, 0);

    // Eingabe von diesem Objekt
    int nAccountNumber;
    float dBalance;
    inp >> nAccountNumber >> dBalance;

    // alloziere einen Puffer und verbinde ihn
    // mit einem ostrstream-Objekt
    char* pszBuffer = new char[128];
    ostrstream out(pszBuffer, 128);

    // Ausgabe an dieses Objekt
   out << »Kontonummer = » << nAccountNumber
       << », Kontostand = $« << dBalance
       << ends;

    return pszBuffer;
}
```

Die Funktion scheint komplizierter zu sein, als sie sein müsste, `parseString( )` ist jedoch einfach zu schreiben aber sehr robust. Die Funktion `parseString( )` kann jeden Typ Input behandeln, den der C++-Extraktor behandeln kann, und sie hat alle Formatierungsfähigkeiten des C++-Inserters. Außerdem ist die Funktion tatsächlich sehr einfach, wenn Sie verstanden haben, was sie tut.

Lassen Sie uns z.B. annehmen, dass `pszString` auf die folgende Zeichenkette zeigt:

```
»1234 100.0«
```

Die Funktion `parseString( )` assoziiert das Objekt `inp` mit der Eingabezeichenkette, indem dieser Wert an den Konstruktor von `istrstream` übergeben wird. Das zweite Argument des Konstruktors ist die Länge der Zeichenkette. In diesem Beispiel ist das Argument gleich 0, was bedeutet »lies bis zum terminierenden Nullzeichen«.

Die Extraktor-Anweisung `inp >>` liest erst die Kontonummer, 1234, in die `int`-Variable `nAccountNumber`, genauso, als wenn sie von der Tastatur oder aus einer Datei gelesen würde. Der zweite Teil liest den Wert 100.0 in die Variable `dBalance`.

Bei der Ausgabe wird das Objekt `out` assoziiert mit dem 128 Zeichen umfassenden Puffer, auf den `pszBuffer` zeigt. Auch hier gibt das zweite Argument die Länge des Puffers an – für diesen Wert kann es keinen Default-Wert geben, weil `ofstream` keine Möglichkeit hat, die Länge des Puffers selber festzustellen (es gibt hier kein abschließendes Nullzeichen). Ein drittes Argument, das dem Modus entspricht, hat `ios::out` als Default-Wert. Sie können dieses Argument jedoch auf `ios::ate` setzen, wenn Sie die Ausgabe an das hängen möchten, was sich bereits im Puffer befindet, anstatt den Puffer zu überschreiben.

Die Funktion gibt dann das out-Objekt aus – das erzeugt die formatierte Ausgabe in den Puffer der 128 Zeichen. Schließlich gibt die Funktion `parseString( )` den Puffer zurück. Die lokal definierten Objekte `inp` und `out` werden bei Rückkehr der Funktion vernichtet.

Die Konstante `ends`, die am Ende des Inserter-Kommandos steht, ist nötig, um den `Null`-Terminator an das Ende der Pufferzeichenkette anzufügen.

Der Puffer, der durch den vorangegangenen Codeschnipsel zurückgegeben wurde, enthält die folgende Zeichenkette:

```
»Kontonummer = 1234, Kontostand = $100.00«
```

29.3.1 Vergleich von Techniken der Zeichenkettenverarbeitung

Die Streamklassen für Zeichenketten stellen ein äußerst mächtiges Konzept dar. Das wird selbst in einem einfachen Beispiel klar. Nehmen Sie an, ich habe eine Funktion, die eine beschreibende Zeichenkette für ein `USDollar`-Objekt erstellen soll.

Meine Lösung ohne Verwendung von `ostrstream` sieht wie in Listing 29-1 aus.

Listing 29-1: Konvertierung von USDollar in eine Zeichenkette zur Ausgabe

```
// ToStringWOStream - konvertiert USDollar in eine
//                    Zeichenkette
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

// USDollar - repräsentiert den US Dollar
class USDollar
{
  public:
    // konstruiere ein USDollar-Objekt mit einem
    // initialen Wert
    USDollar(int d = 0, int c = 0);
```

```
// rationalize - normalisiere nCents durch
    //              Addition eines Dollars pro
    //              100 Cents
    void rationalize()
    {
        nDollars += (nCents / 100);
        nCents   %= 100;
    }

    // output - gib eine Beschreibung des aktuellen
    //          Objektes zurück
    char* output();

  protected:
    int nDollars;
    int nCents;
};

USDollar::USDollar(int d, int c)
{
    // speichere die initialen Werte
    nDollars = d;
    nCents = c;

    rationalize();
}

// output - gib eine Beschreibung des aktuellen
//          Objektes zurück
char* USDollar::output()
{
    // alloziere einen Puffer
    char* pszBuffer = new char[128];

    // konvertiere den Wert von nDollar und nCents
    // in Zeichenketten
    char cDollarBuffer[128];
    char cCentsBuffer[128];
    ltoa((long)nDollars, cDollarBuffer, 10);
    ltoa((long)nCents,   cCentsBuffer,  10);

    // Cents sollen 2 Ziffern benutzen
    if (strlen(cCentsBuffer) != 2)
    {
        char c = cCentsBuffer[0];
        cCentsBuffer[0] = '0';
        cCentsBuffer[1] = c;
        cCentsBuffer[2] = '\0';
    }

    // füge die Zeichenketten zusammen

    strcpy(pszBuffer, »$«);
    strcat(pszBuffer, cDollarBuffer);
    strcat(pszBuffer, ».«);
```

```
    strcat(pszBuffer, cCentsBuffer);
    return pszBuffer;
}

int main(int nArgc, char* pszArgs[])
{
    USDollar d1(1, 60);
    char* pszD1 = d1.output();
    cout << »Dollar d1 = » << pszD1 << »\n«;
    delete pszD1;

    USDollar d2(1, 5);
    char* pszD2 = d2.output();
    cout << »Dollar d2 = » << pszD2 << »\n«;
    delete pszD2;

    return 0;
}
```

Ausgabe

```
Dollar d1 = $1.60
Dollar d2 = $1.05
```

Das Programm ToStringWOStream stützt sich nicht auf die Streamroutinen, um den Text für das `USDollar`-Objekt zu erzeugen. Die Funktion `USDollar::output( )` macht intensiven Gebrauch von der Funktion `ltoa( )`, die long in eine Zeichenkette verwandelt, und von den Funktionen `strcpy( )` und `strcat( )`, die direkte Manipulationen auf Zeichenketten ausführen. Die Funktionen müssen selber mit dem Fall zurecht kommen, dass die Anzahl Cents kleiner als 10 ist und daher nur eine Stelle belegt. Die Ausgabe des Programms finden Sie am Ende des Listings.

Das Folgende zeigt eine Version von `USDollar::output( )`, die die Klasse `ostrstream` verwendet.

```
char* USDollar::output()
{
    // alloziere einen Puffer
    char* pszBuffer = new char[128];

    // verbinde einen ostream mit dem Puffer
    ostrstream out(pszBuffer, 128);

    // konvertiere in Zeichenketten (Setzen von
    // width stellt sicher, dass die Breite der
    // Cents nicht kleiner als 2 ist)
    out << »$« << nDollars << ».«;
    out.fill('0');
    out.width(2);
    out << nCents << ends;

    return pszBuffer;
}
```

Diese Version ist im Programm ToStringWStreams auf der beiliegenden CD-ROM enthalten.

Diese Version assoziiert den Ausgabestream `out` mit einem lokal definierten Puffer. Sie schreibt dann die nötigen Werte unter Verwendung der üblichen Stream-Inserter und gibt den Puffer zurück. Das Setzen der Breite auf 2 stellt sicher, dass die Anzahl der verwendeten Stellen auch dann zwei ist, wenn der Wert kleiner als 10 ist. Die Ausgabe dieser Version ist identisch mit der Ausgabe von Listing 29-1. Das `out`-Objekt wird vernichtet, wenn die Kontrolle die Funktion `output( )` verlässt.

Ich finde, dass die Stream-Version von `output( )` viel besser verfolgt werden kann und weniger langweilig ist als die frühere Version, die keine Streams nutzte.

29.4 Manipulatoren

Bis jetzt haben wir gesehen, wie Stream-I/O verwendet werden kann, um Zahlen und Zeichenkette auszugeben unter Verwendung von Default-Formaten. Normalerweise sind die Defaults in Ordnung, aber manchmal treffen sie es einfach nicht. Weil dies so ist, stellt C++ zwei Wege bereit, um die Formatierung der Ausgabe zu kontrollieren.

Erstens kann der Aufruf einer Reihe von Elementfunktionen des Stream-Objektes das Format steuern. Sie haben das in einer früheren Elementfunktion `display( )` gesehen, in der `fill('0')` und `width(2)` die minimale Breite und das Linksfüllzeichen eines `ostrstream`-Objektes gesetzt haben.

Das Argument `out` ***stellt ein*** `ostream`***-Objekt dar. Weil*** `ostream` ***Basisklasse für*** `ofstream` ***und*** `ostrstream` ***ist, funktioniert die Funktion gleich gut für die Ausgabe in eine Datei und einen im Programm bereitgestellten Puffer.***

Ein zweiter Zugang ist der über *Manipulatoren*. Manipulatoren sind Objekte, die in der Include-Datei iomanip.h definiert sind, die den gleichen Effekt haben wie Aufrufe von Elementfunktionen.

Der einzige Vorteil von Manipulatoren ist, dass das Programm sie direkt in den Stream einfügen kann und keinen separaten Funktionsaufruf ausführen muss.

Die Funktion `display( )` kann mit Manipulatoren wie folgt umgeschrieben werden:

```
char* USDollar::output()
{
    // alloziere einen Puffer
    char* pszBuffer = new char[128];

    // verbinde einen ostream mit dem Puffer
    ostrstream out(pszBuffer, 128);

    // konvertiere in Zeichenketten; diese Version
    // verwendet Manipulatoren zum Setzen des
```

```
    // Füllzeichens und der Breite
    out << »$« << nDollars << ».«
        << setfill('0') << setw(2)
        << nCents << ends;

    return pszBuffer;
}
```

Die geläufigsten Manipulatoren und ihre Bedeutung finden Sie in Tabelle 29-4.

Tabelle 29-4: Manipulatoren und Elementfunktionen zur Formatkontrolle

Manipulator	Elementfunktion	Beschreibung
`dec`	`flags(10)`	Setze Radix auf 10
`hex`	`flags(16)`	Setze Radix auf 16
`Oct`	`flags(8)`	Setze Radix auf 8
`setfill(c)`	`fill(c)`	Setze Füllzeichen auf `c`
`setprecision(c)`	`precision(c)`	Setze Genauigkeit auf `c`
`setw(n)`	`width(n)`	Setze Feldbreite auf `n` Zeichen

Sehen Sie nach dem Breitenparameter (Funktion `width( )` und Manipulator `setw( )`). Die meisten Parameter behalten ihren Wert, bis sie durch einen weiteren Aufruf neu gesetzt werden, aber der Breitenparameter verhält sich so nicht. Der Breitenparameter wird auf seinen Default-Wert gesetzt, sobald die nächste Ausgabe erfolgt. Sie könnten z.B. von dem Folgenden erwarten, dass Integerzahlen mit acht Zeichen erzeugt werden:

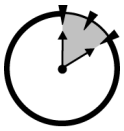
```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
   cout << setw(8)      // Breite ist 8...
        << 10           // ... für die 10, aber...
        << 20           // ... default für die 20
        << »\n«;
}
```

Was Sie jedoch erhalten, ist eine Integerzahl mit acht Zeichen, gefolgt von einer Integerzahl mit zwei Zeichen. Um auch für die zweite Zahl acht Zeichen zu bekommen, ist das Folgende notwendig:

```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
   cout << setw(8)      // setze die Breite ...
        << 10
        << setw(8)      // ... und setze sie wieder
        << 20
```

```
        << »\n«;
}
```

Was ist besser, Manipulatoren oder Aufrufe von Elementfunktionen? Elementfunktionen erlauben etwas mehr Kontrolle, weil es mehr davon gibt. Außerdem geben die Elementfunktionen die vorherigen Einstellungen zurück, was Sie nutzen können, um die Werte wieder zurückzusetzen, wenn Sie das möchten. Schließlich hat jede Funktion eine Version ohne Argumente, um den aktuellen Wert zurückzugeben, falls Sie die Einstellungen später wieder zurücksetzen möchten.



10 Min.

29.5 Benutzerdefinierte Inserter

Die Tatsache, dass C++ den Linksshiftoperator überlädt, um Ausgaben auszuführen, ist praktisch, weil Sie dadurch die Möglichkeit bekommen, denselben Operator für die Ausgabe der von Ihnen definierten Klassen zu überladen.

Betrachten Sie die Klasse `USDollar` noch einmal. Die folgende Version der Klasse enthält einen Inserter, der die gleiche Ausgabe erzeugt wie die frühere Version von `display( )`:

```
// Inserter - stellt einen Inserter für USDollar
//           bereit
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>

// USDollar - repräsentiert den US Dollar
class USDollar
{
    friend ostream& operator<<(ostream& out, USDollar& d);
  public:
    // ... keine Änderungen ...
};

// Inserter - Ausgabe als Zeichenkette
//           (diese Version behandelt den Fall, dass
//           die Cents kleiner als 10 sind)
ostream& operator<<(ostream& out, USDollar& d)
{
    char old = out.fill();
    out << »$«
        << d.nDollars
        << ».«
        << setfill('0') << setw(2)
        << d.nCents;

    // setze wieder das alte Füllzeichen
    out.fill(old);

    return out;
}

int main(int nArgc, char* pszArgs[])
{
    USDollar d1(1, 60);
```

```
    cout << »Dollar d1 = » << d1 << »\n«;

    USDollar d2(1, 5);
    cout << »Dollar d2 = » << d2 << »\n«;

    return 0;
}
```

Der Inserter führt die gleichen elementaren Operationen aus wie die frühere Funktion `display( )`, wobei hier direkt in das `ostream`-Ausgabeobjekt ausgegeben wird, das übergeben wurde. Die Funktion `main( )` ist jedoch noch einfacher als vorher. Dieses Mal kann das `USDollar`-Objekt direkt in den Ausgabestream eingefügt werden.

Sie wundern sich vielleicht, warum `operator<<( )` das `ostream`-Objekt zurückgibt, das übergeben wurde. Der Grund ist, dass dadurch Einfügeoperationen verkettet werden können. Weil `operator<<( )` von links nach rechts bindet, wird der folgende Ausdruck

```
USDollar d1(1, 60);
cout << »Dollar d1 = » << d1 << »\n«;
```

interpretiert als

```
USDollar d1(1, 60);
((cout << »Dollar d1 = ») << d1) << »\n«;
```

Die erste Eingabe gibt die Zeichenkette »Dollar d1 = « nach `cout` aus. Das Ergebnis dieses Ausdrucks ist das Objekt `cout`, das dann an `operator<<(ostream&, USDollar&)` übergeben wird. Es ist wichtig, dass dieser Operator sein `ostream`-Objekt zurückgibt, so dass dieses Objekt an den nächsten Inserter übergeben werden kann, der das Zeilenendezeichen »\n« ausgibt.

29.6 Schlaue Inserter

Wir möchten die Inserter oft schlau machen. D.h. wir möchten gerne schreiben `cout << baseClassObjekt`, und dann C++ den passenden Inserter einer Unterklasse wählen lassen in der gleichen Weise, wie C++ die richtige virtuelle Elementfunktion gewählt hat. Weil der Inserter keine Elementfunktion ist, können wir ihn nicht direkt als virtual deklarieren.

Wir können das Problem leicht umgehen, indem wir den Inserter von einer virtuellen Funktion `display( )` abhängig machen, wie im Programm `VirtualInserter` in Listing 29-2 gezeigt wird.

Listing 29-2: Programm VirtualInserter

```
// VirtualInserter - basiere USDollar auf einer
//                   Basisklasse Currency (Währung);
//                   mache den Inserter virtual,
//                   indem er sich auf die Methode
//                   display() stützt
#include <stdio.h>
#include <iostream.h>
#include <iomanip.h>

// Currency - stellt Währung dar
class Currency
```

```
{
    friend ostream& operator<<(ostream& out, Currency& d);
  public:
    Currency(int p = 0, int s = 0)
    {
        nPrimary = p;
        nSecondary = s;
    }

    // rationalize - normalisiere nSecondary durch
    //               Inkrementieren von nPrimary
    //               pro 100 in nSecondary
    void rationalize()
    {
        nPrimary   += (nSecondary / 100);
        nSecondary %= 100;
    }

    // display - Schreiben des Objektes in das
    //           gegebene ostream-Objekt
    virtual ostream& display(ostream&) = 0;

  protected:
    int nPrimary;
    int nSecondary;
};

// Inserter - Ausgabe als Zeichenkette
//            (diese Version behandelt den Fall, dass
//            nSecondary kleiner als 10 sind)
ostream& operator<<(ostream& out, Currency& c)
{
    return c.display(out);
}

// definiere USDollar als Unterklasse von Currency
class USDollar : public Currency
{
  public:
    USDollar(int d, int c) : Currency(d, c)
    {
    }

    // Ausgaberoutine
    virtual ostream& display(ostream& out)
    {
        char old = out.fill();
        out << »$«
            << nPrimary
            << ».«
            << setfill('0') << setw(2)
            << nSecondary;

        // altes Füllzeichen aktivieren
        out.fill(old);
```

```
        return out;
    }
};

void fn(Currency& c, char* pszDescriptor)
{
    cout << pszDescriptor << c << »\n«;
}

int main(int nArgc, char* pszArgs[])
{
   // rufe USDollar::display() direkt auf
   USDollar d1(1, 60);
   cout << »Dollar d1 = » << d1 << »\n«;

   // rufe die gleiche Funktion virtuell auf
   // über die Funktion fn()
   USDollar d2(1, 5);
   fn(d2, »Dollar d2 = »);

   return 0;
}
```

Die Klasse `Currency` definiert eine Inserter-Funktion, die ein Nichtelement ist, und daher mit Polymorphie nichts zu tun hat. Aber statt wirklich etwas zu tun, stützt sich der Inserter auf eine virtuelle Elementfunktion `display( )`, die die eigentliche Arbeit ausführt. Die Unterklasse `USDollar` muss nur die Funktion `display( )` bereitstellen; das ist alles. Diese Version des Programms erzeugt die gleiche Ausgabe wie am Ende von Listing 29-1 zu sehen ist.

Dass die Einfügeoperation in der Tat polymorph ist, wird bei der Erzeugung der Ausgabefunktion `fn(Currency&, char*)` deutlich. Die Funktion `fn( )` kennt nicht den Typ der Währung, den sie übergeben bekommt, und stellt die übergebene Währung mit den für `USDollar` geltenden Regeln dar. `main( )` gibt `d1` direkt und `d2` über diese neue Funktion `fn( )` aus. Die virtuelle Ausgabe von `fn( )` sieht genauso aus, wie die des polymorphen Bruders.

Andere Unterklassen von `Currency`, wie z.B. `DMark`, `FFranc` oder `Euro`, können erzeugt werden, obwohl sie unterschiedliche Darstellungsregeln haben, indem einfach die entsprechenden `display( )`-Funktionen bereitgestellt werden. Der Basiscode kann weiterhin ungestraft `Currency` verwenden.

29.7 Aber warum die Shift-Operatoren?

Sie können fragen »Warum die Shift-Operatoren für Stream-I/O verwenden? Warum nicht einen anderen Operator?«

Der Linksshift-Operator wurde aus mehreren Gründen gewählt. Erstens ist er ein binärer Operator. Das bedeutet, dass das `ostream`-Objekt das Argument auf der linken Seite, und das Ausgabeobjekt das Argument auf der rechten Seite sein kann. Zweitens ist der Links-Shift ein Operator auf einer niedrigen Ebene. Somit arbeiten Ausdrücke wie der folgende wie erwartet, weil Addition vor dem Einfügen ausgeführt wird:

```
cout << »a + b« << a + b << »\n«;
```

Drittens bindet der Linksshiftoperator von links nach rechts. Das erlaubt es uns, Ausgabeanweisungen zu verketten. Die vorige Zeile wird z.B. interpretiert als:

```
((cout << »a + b«) << a + b) << »\n«;
```

0 Min.

Trotz all dieser Gründe ist der eigentliche Grund sicherlich, dass es schön ist. Das doppelte Kleinerzeichen << sieht so aus, als wenn etwas den Code verlassen wollte, und das doppelte Größerzeichen >> sieht so aus, als wenn etwas hereinkommen wollte. Und, warum eigentlich nicht?

Zusammenfassung

Ich habe diese Sitzung mit einer Warnung begonnen, dass Stream-I/O zu komplex ist, um in einem Kapitel eines Buches abgehandelt zu werden. Sie können die Dokumentation Ihres Compilers bemühen, um eine vollständige Liste aller Elementfunktionen zu erhalten, die sie aufrufen können. Die relevanten Include-Dateien, wie iostream.h und iomanip.h, enthalten Prototypen mit erklärenden Kommentaren für alle Funktionen.

- Stream-I/O basiert auf den Klassen `istream` und `ostream`.
- Die Include-Datei `iostream.h` überlädt den Linksshift-Operator, um Ausgaben nach `ostream` auszuführen und überlädt den Rechtsshift-Operator, um Eingaben von `istream` auszuführen.
- Die Unterklasse `fstream` wird für Datei-I/O verwendet.
- Die Unterklasse `strstream` führt I/O auf internen Speicherpuffern durch, unter Verwendung der gleichen Einfüge- und Extraktionsoperatoren.
- Der Programmierer kann die Einfüge- und Extraktionsoperatoren überladen für seine eigenen Klassen. Diese Operatoren können polymorph gemacht werden durch die Verwendung virtueller Zwischenmethoden.
- Die Manipulatorobjekte, die in iomanip.h definiert werden, können verwendet werden, um Formatfunktionen von `stream` aufzurufen.

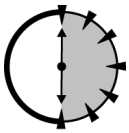
Selbsttest

1. Wie werden die beiden Operatoren << und >> genannt, wenn sie für Stream-I/O verwendet werden? (Siehe »Wie funktioniert Stream-I/O?«)
2. Was ist die Basisklasse der beiden Default-I/O-Objekte `cout` und `cin`? (Siehe »Wie funktioniert Stream-I/O?«)
3. Wofür wird die Klasse `fstream` verwendet? (Siehe »Die Unterklassen fstream«)
4. Wofür wird die Klasse `strstream` verwendet? (Siehe »Die Unterklassen strstream«)
5. Welcher Manipulator setzt den nummerischen Ausgabemode auf hexadezimal? Was ist die zugehörige Elementfunktion? (Siehe »Manipulatoren«)

Ausnahmen

Checkliste

- ☑ Fehlerbedingungen zurückgeben
- ☑ Ausnahmen verwenden, ein neuer Mechanismus zur Fehlerbehandlung
- ☑ Auslösen und Abfangen von Ausnahmen
- ☑ Überladen der Ausnahmeklasse



30 Min.

Zusätzlich zu dem allgegenwärtigen Ansatz der Fehlerausgaben enthält C++ einen einfacheren und verlässlicheren Mechanismus zur Fehlerbehandlung. Diese Technik, die Ausnahmebehandlung genannt wird, ist der Gegenstand dieser Sitzung.

30.1 Konventionelle Fehlerbehandlung

Eine Implementierung des allgemeinen Beispiels der Fakultät sieht folgendermaßen aus.

Die Fakultätsfunktion finden Sie auf der beiliegenden CD-ROM im Programm FactorialProgram.cpp.

```
// factorial - berechne die Fakultät von nBase, die
//             gleich nBase * (nBase - 1) *
//             (nBase - 2) * ... ist
int factorial(int nBase)
{
    // starte mit Wert 1
    int nFactorial = 1;

    // Schleife von nBase bis 1, jedes Mal den
    // Produktwert mit dem aktuellen Wert
    // multiplizieren
```

```
    do
    {
        nFactorial *= nBase;
    } while (--nBase > 1);

    // return the result
    return nFactorial;
}
```

Obwohl die Funktion sehr einfach ist, fehlt ihr ein kritisches Feature: Die Fakultät von 0 ist 1, während die Fakultät einer negativen Zahl nicht definiert ist. Die obige Funktion sollte einen Test enthalten für negative Argumente und eine Fehlermeldung ausgeben, falls ein solches übergeben wird.

Der klassische Weg, einen Fehler in einer Funktion anzuzeigen, ist die Rückgabe eines Wertes, der sonst nicht von der Funktion zurückgegeben werden kann. Z.B. ist es nicht möglich, dass die Fakultät negativ ist. Wenn der Funktion also ein negativer Wert übergeben wird, könnte sie z.B. -1 zurückgeben. Die aufrufende Funktion kann den Rückgabewert überprüfen – wenn er negativ ist, weiß die aufrufende Funktion, dass ein Fehler aufgetreten ist und kann eine entsprechende Aktion auslösen (was immer das dann ist).

Das ist die Art und Weise der Fehlerbehandlung, die seit den frühen Tagen von FORTRAN praktiziert wurde. Warum sollte das geändert werden?

30.2 Warum benötigen wir einen neuen Fehlermechanismus?

Es gibt verschiedene Probleme mit dem Ansatz der Fehlerausgaben. Zum einen ist nicht jede Funktion in der glücklichen Lage wie die Fakultätsfunktion, dass keine negativen Werte zurückkommen können. Nehmen Sie z.B. den Logarithmus. Sie können den Logarithmus nicht für eine negative Zahl berechnen, aber der Logarithmus kann positiv und negativ sein – es gibt keinen Wert, der von einer Funktion `logarithm( )` zurückgegeben werden könnte, der nicht ein gültiger Logarithmuswert ist.

Zweitens gibt es zu viele Informationen, die in einem Integerwert gespeichert werden müssten. Z.B. -1 für »Argument ist negativ« und -2 für »Argument zu groß«, aber wenn das Argument zu groß ist, gibt es keine Möglichkeit, das Ergebnis zurückzugeben. Die Kenntnis dieses Wertes würde aber vielleicht zur Lösung des Problems beitragen. Es gibt keine Möglichkeit, als nur einen einzigen Rückgabewert zu speichern.

Drittens ist die Behandlung von Fehlern optional. Nehmen Sie an, jemand schreibt `factorial()` so, dass sie die Argumente überprüft und einen negativen Wert zurückgibt, wenn das Argument negativ ist. Wenn der Code, der diese Funktion benutzt, den Rückgabewert nicht überprüft, hilf das gar nichts. Natürlich sprechen wir alle möglichen Drohungen aus »Sie werden Ihre Fehlerrückgaben überprüfen oder ...« aber wir wissen alle, dass die Sprache (und Ihr Chef) nichts tun kann, wenn Sie es unterlassen.

Selbst wenn ich die Fehlerrückgabe von `factorial( )` oder einer anderen Funktion überprüfe, was kann meine Funktion mit dem Fehler anfangen? Sicherlich nicht mehr, als eine Fehlermeldung auszugeben oder selber einen Fehlercode an die aufrufende Funktion zurückzugeben. Schnell sieht der Code dann so aus:

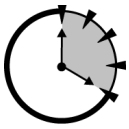
```
// rufe eine Funktion auf, überprüfe den
// Fehlerrückgabewert, behandle ihn und kehre zurück
int nErrorRtn = someFunc();
if (nErrorRtn)
{
    errorOut(»Fehler beim Aufruf von someFunc()«);
    return MY_ERROR_1;
}

nErrorRtn = someOtherFunc();
if (nErrorRtn)
{
    errorOut(»Fehler beim Aufruf von someOtherFunc()«);
    return MY_ERROR_2;
}
```

Dieser Mechanismus hat mehrere Probleme:

- Er wiederholt vieles.
- Er zwingt den Benutzer, verschiedene Fehlermeldungen zu erfinden und abzufangen.
- Er mischt den Code zur Fehlerbehandlung und den normalen Codefluss, wodurch beide unleserlicher werden.

Diese Probleme scheinen in diesem einfachen Beispiel nicht so schlimm zu sein, aber die Komplexität nimmt rapide zu, wenn die Komplexität des aufrufenden Codes zunimmt. Nach einer Weile gibt es mehr Code zur Fehlerbehandlung als »eigentlichen« Code.

Das Ergebnis ist, dass Code zur Fehlerbehandlung nicht so geschrieben wird, dass alle Bedingungen erfasst sind, die erfasst werden müssen.



20 Min.

30.3 Wie arbeiten Ausnahmen?

C++ führt einen total neuen Mechanismus zum Abfangen und Behandeln von Fehlern ein. Dieser Mechanismus wird als Ausnahmen bezeichnet und basiert auf den Schlüsselworten `try`, `throw` und `catch`. Er arbeitet etwa so: eine Funktion versucht (`try`), durch ein Stück Code hindurch zu kommen. Wenn der Code ein Problem entdeckt, löst es einen Fehler aus (`throw`), der von der Funktion abgefangen werden kann (`catch`).

Listing 30-1 zeigt, wie Ausnahmen arbeiten.

Listing 30-1: Ausnahmen in Aktion

```
// FactorialExceptionProgram - Ausgabe der Fakultät
//                           mit Ausnahme-basierter
//                           Fehlerbehandlung
#include <stdio.h>
#include <iostream.h>

// factorial - berechne die Fakultät von nBase, die
//             gleich nBase * (nBase - 1) *
//             (nBase - 2) * ... ist
int factorial(int nBase)
```

```
{
    // wenn nBase < 0...
    if (nBase <= 0)
    {
        // ... löse einen Fehler aus
        throw »Ausnahme ungültiges Argument«;
    }

    int nFactorial = 1;
    do
    {
        nFactorial *= nBase;
    } while (—nBase > 1);

    // return the result
    return nFactorial;
}

int main(int nArgc, char* pszArgs[])
{
    // rufe factorial in einer Schleife auf,
    // fange alle Ausnahmen ab, die die Funktion
    // auslösen könnte
    try
    {
        for (int i = 6; ; i—)
        {
            cout << »factorial(»
                 << i
                 << ») = »
                 << factorial(i)
                 << »\n«;
        }
    }
    catch(char* pErrorMsg)
    {
        cout << »Fehler: »
             << pErrorMsg
             << »\n«;
    }
    return 0;
}
```

Die Funktion `main( )` beginnt mit einem Block, der mit dem Schlüsselwort `try` markiert ist. Einer oder mehr `catch`-Blöcke stehen unmittelbar hinter dem `try`-Block. Das Schlüsselwort `try` wird von einem einzigen Argument gefolgt, das wie eine Funktionsdefinition aussieht.

Innerhalb des `try`-Blocks kann `main( )` tun, was sie will. In diesem Fall geht `main( )` in eine Schleife, die die Fakultät von absteigenden Zahlen berechnet. Schließlich übergibt das Programm eine negative Zahl an die Funktion `factorial( )`.

Wenn unsere schlaue Funktion `factorial( )` eine solche falsche Anfrage bekommt, »wirft« sie eine Zeichenkette, die eine Beschreibung des Fehlers enthält, unter Verwendung des Schlüsselwortes `throw`.

An diesem Punkt sucht C++ nach einem `catch`-Block, dessen Argument zu dem ausgelösten Objekt passt. Der Rest des `try`-Blocks wird nicht fertig abgearbeitet. Wenn C++ kein `catch` in der aktuellen Funktion findet, kehrt C++ zum Ausgangspunkt des Aufrufes zurück und setzt die Suche dort fort. Der Prozess wird fortgesetzt, bis ein passender `catch`-Block gefunden wird oder die Kontrolle `main( )` verlässt.

In diesem Beispiel wird die ausgelöste Fehlermeldung durch den `catch`-Block am Ende der Funktion `main( )` abgefangen, der eine Meldung ausgibt. Die nächste Anweisung ist das return-Kommando, wodurch das Programm beendet wird.

30.3.1 Warum ist der Ausnahmemechanismus eine Verbesserung?

Der Ausnahmemechanismus löst die Probleme, die dem Mechanismus der Fehlerausgaben inhärent sind, indem der Pfad zur Fehlerbehandlung vom Pfad des normalen Codes getrennt wird. Außerdem machen Ausnahmen die Fehlerbehandlung zwingend. Wenn Ihre Funktion die ausgelöste Ausnahme nicht verarbeitet, geht die Kontrolle die Kette der aufrufenden Funktionen nach oben, bis C++ eine Funktion findet, die diese Ausnahme behandeln kann. Das gibt Ihnen auch die Flexibilität, Fehler zu ignorieren, bei denen Sie nichts machen können. Nur die Funktionen, die das Problem beheben können, müssen die Ausnahme abfangen. Und wie funktioniert das?

30.4 Abfangen von Details, die für mich bestimmt sind

Lassen Sie uns die Schritte genauer ansehen, die der Code durchlaufen muss, um eine Ausnahme zu verarbeiten. Wenn ein throw ausgeführt wird, kopiert C++ das ausgelöste Objekt an einen neutralen Ort. Dann wird das Ende des `try`-Blocks untersucht. C++ sucht nach einem passenden `catch` irgendwo in der Kette der Funktionsaufrufe. Dieser Prozess wird »Abwickeln des Stacks« genannt.

Ein wichtiges Feature des Stackabwickelns ist, dass jedes Objekt, das seine Gültigkeit verliert, vernichtet wird, genauso als wenn die Funktion eine return-Anweisung ausgeführt hätte. Das verhindert, dass das Programm Ressourcen verliert, oder Objekte »in der Luft hängen«.

Listing 30-2 ist ein Beispielprogramm, das den Stack abwickelt:

Listing 30-2: Abwickeln des Stacks

```
#include <iostream.h>
class Obj
{
  public:
    Obj(char c)
    {
        label = c;
        cout << »Konstruktor » << label << endl;
    }
   ~Obj()
    {
        cout << »Destruktor »  << label << endl;
    }

  protected:
    char label;
};
void f1();
```

```
void f2();

int main(int, char*[])
{
    Obj a('a');
    try
    {
        Obj b('b');
        f1();
    }
    catch(float f)
    {
        cout << »float abgefangen« << endl;
    }
    catch(int i)
    {
        cout << »int abgefangen« << endl;
    }
    catch(...)
    {
        cout << »generisches catch« << endl;
    }
    return 0;
}

void f1()
{
    try
    {
        Obj c('c');
        f2();
    }
    catch(char* pMsg)
    {
        cout << »Zeichenkette abgefangen« << endl;
    }
}

void f2()
{
    Obj d('d');
    throw 10;
}
```

Ausgabe:

```
Konstruktor a
Konstruktor b
Konstruktor c
Konstruktor d
Destruktor d
Destruktor c
Destruktor b
int abgefangen
Destruktor a
```

Zuerst werden die vier Objekte `a`, `b`, `c` und `d` konstruiert, wenn die Kontrolle ihre Deklarationen antrifft, bevor `f2( )` die `int` 10 auslöst. Weil kein `try`-Block definiert ist innerhalb von `f2( )`, wickelt C++ den Stack von `f2` ab, was zur Vernichtung von `d` führt. `f1( )` definiert einen `try`-Block, aber ihr einziger `catch`-Block verarbeitet `char*`, und passt daher nicht zu dem ausgelösten `int`, und C++ setzt die Suche fort. Das wickelt den Stack von `f1( )` ab, wodurch das Objekt `c` vernichtet wird.

Zurück in `main( )` findet C++ einen weiteren `try`-Block. Das Verlassen dieses Block vernichtet `b`. Der erste `catch`-Block verarbeitet `float`, und passt daher nicht. Der nächste `catch`-Block passt exakt. Der letzte `catch`-Block, der jedes beliebige Objekt verarbeiten würde, wird nicht mehr betreten, weil bereits ein passender `catch`-Block gefunden wurde.

Eine Funktion, die als `fn(...)` ***deklariert ist, akzeptiert eine beliebige Anzahl von Argumenten mit beliebigem Typ. Das gleiche gilt für*** `catch`***-Blöcke. Ein*** `catch(...)` ***fängt alles ab.***

10 Min.

30.4.1 Was kann ich werfen?

Ein C++-Programm kann jeden beliebigen Typ von Objekten auslösen. C++ verwendet einfache Regeln, um einen passenden `catch`-Block zu finden.

C++ untersucht zuerst die `catch`-Blöcke, die direkt hinter dem `try`-Block stehen. Die `catch`-Blöcke werden der Reihe nach betrachtet, bis ein Block gefunden wurde, der zu dem ausgelösten Objekt passt. Ein »Treffer« wird mit den gleichen Regeln definiert, wie ein Treffer für Argumente in einer überladenen Funktion. Wenn kein passender `catch`-Block gefunden wurde, geht der Code zu den `catch`-Blocks im nächsthöheren Level, wie auf einer Spirale nach oben, bis ein passender `catch`-Block gefunden wird. Wenn kein `catch`-Block gefunden wird, beendet sich das Programm.

Die frühere Funktion `factorial( )` wirft eine Zeichenkette, d.h. ein Objekt von Typ `char*`. Die zugehörige `catch`-Deklaration passt, weil sie verspricht, ein Objekt vom Typ `char*` zu verarbeiten. Eine Funktion kann jedoch ein Objekt eines beliebigen Typs auslösen.

Ein Programm kann das Ergebnis eines Ausdrucks auslösen. Das bedeutet, dass Sie so viel Information mitgeben können, wie sie möchten. Betrachten Sie die folgende Klassendefinition:

```
#include <iostream.h>
#include <string.h>
// Exception - generische Klasse für Fehlerbehandlung
class Exception
{
  public:
    // konstruiere ein Ausnahmeobjekt mit einem
    // Beschreibungstext des Problems, zusammen mit
    // der Datei und der Zeilennummer, wo das
    // Problem aufgetreten ist
    Exception(char* pMsg, char* pFile, int nLine)
    {
        this->pMsg = new char[strlen(pMsg) + 1];
        strcpy(this->pMsg, pMsg);

        strncpy(file, pFile, sizeof file);
```

```
        file[sizeof file - 1] = '\0';
        lineNum = nLine;
    }

    // display - Ausgabe des Inhalts des aktuellen
    //           Objektes in den Ausgabestream
    virtual void display(ostream& out)
    {
        out << »Fehler <« << pMsg << »>\n«;
        out << »in Zeile #«
            << lineNum
            << », in Datei »
            << file
            << endl;
    }

  protected:
    // Fehlermeldung
    char* pMsg;

    // Dateiname und Zeilennummer des Fehlers
    char file[80];
    int lineNum;
};
Das entsprechende throw sieht wie folgt aus:
throw Exception(»Negatives Argument für factorial«,
                __FILE__,
                __LINE__);
```

__FILE__ und __LINE__ sind elementare #defines, die auf den Namen der Quelldatei und die aktuelle Zeile in der Datei gesetzt sind.

Der zugehörige catch-Block sieht so aus:

```
void myFunc()
{
    try
    {
        //... was auch immer aufgerufen wird
    }

     // fange ein Exception-Objekt ab
     catch(Exception x)
     {
          // verwende die eingebaute Elementfunktion
          // display zur Anzeige des Objektes im
          // Fehlerstream
          x.display(cerr);
     }
}
```

Der `catch`-Block nimmt das `Exception`-Objekt und verwendet dann die eingebaute Elementfunktion `display( )` zur Anzeige der Fehlermeldung.

Die Version von factorial, die von der Klasse `Exception` ***Gebrauch macht, ist auf der beiliegenden CD-ROM als FactorialThrow.cpp enthalten.***

Die Ausgabe bei Ausführung des Fakultätsprogramms sieht so aus:

```
factorial(6) = 720
factorial(5) = 120
factorial(4) = 24
factorial(3) = 6
factorial(2) = 2
factorial(1) = 1
Error <Negatives Argument für factorial>
in Zeile #59,
in Datei C:\wecc\Programs\lesson30\FactorialThrow.cpp
```

Die Klasse `Exception` stellt eine generische Klasse zum Melden von Fehlern dar. Sie können Unterklassen von dieser Klasse ableiten. Ich könnte z.B. eine Klasse `InvalidArgumentException` ableiten, die den unzulässigen Argumentwert speichert, zusammen mit dem Fehlertext und dem Ort des Fehlers.

```
class InvalidArgumentException : public Exception
{
  public:
    InvalidArgumentException(int arg, char* pFile,
                             int nLine)
      : Exception(»Unzulässiges Argument«, pFile, nLine)
    {
        invArg = arg;
    }

    // display - Ausgabe des Objektes in das
    //           angegebene Ausgabeobjekt
    virtual void display(ostream& out)
    {
        // die Basisklasse gibt ihre
        // Informationen aus ...
        Exception::display(out);

        // ... und dann sind wir dran
        out << »Argument war »
            << invArg
            << »\n«;
    }

  protected:
    int invArg;
};
```

Die aufrufende Funktion behandelt die neue Klasse `InvalidArgumentException` automatisch, weil `InvalidArgumentException` eine `Exception` ist und die Elementfunktion `display( )` polymorph ist.

Die Funktion `InvalidArgumentException::display( )` ***stützt sich auf die Basisklasse*** `Exception`***, um den Teil des Objektes auszugeben, der von Exception stammt.***

30.5 Verketten von catch-Blöcken

Ein Programm kann seine Flexibilität in der Fehlerbehandlung dadurch erhöhen, indem mehrere `catch`-Blöcke an den gleichen `try`-Block angefügt werden. Der folgende Codeschnipsel demonstriert das Konzept:

```
void myFunc()
{
    try
    {
        // ... was auch immer aufgerufen wird
    }
    // fange eine Zeichenkette ab
    catch(char* pszString)
    {
        cout << »Fehler: » << pszString << »\n«;
    }
    // fange ein Exception-Objekt ab
    catch(Exception x)
    {
        x.display(cerr);
    }

    // ... Ausführung wird hier fortgesetzt ...
}
```

In diesem Beispiel wird ein ausgelöstes Objekt, wenn es eine einfache Zeichenkette ist, vom ersten `catch`-Block abgefangen, der die Zeichenkette ausgibt. Wenn das Objekt keine Zeichenkette ist, wird es mit der `Exception`-Klasse verglichen. Wenn das Objekt eine `Exception` ist oder aus einer Unterklasse von `Exception` stammt, wird es vom zweiten `catch`-Block verarbeitet.

Weil sich dieser Prozess seriell vorgeht, muss der Programmierer mit den spezielleren Objekttypen anfangen und bei den allgemeineren aufhören. Es ist daher ein Fehler, das Folgende zu tun:

```
void myFunc()
{
    try
    {
        // ... was auch immer aufgerufen wird
    }
    catch(Exception x)
    {
        x.display(cerr);
```

```
    }
    catch(InvalidArgumentException x)
    {
        x.display(cerr);
    }
}
```

Weil eine `InvalidArgumentException` eine `Exception` ist, wird die Kontrolle den zweiten `catch`-Block nie erreichen.

0 Min.

Der Compiler fängt diesen Programmierfehler nicht ab.

Es macht im obigen Beispiel auch keinen Unterschied, dass `display( )` ***virtuell ist. Der catch-Block für*** `Exception` ***ruft die Funktion*** `display( )` ***in Abhängigkeit vom Laufzeittyp des Objektes auf.***

Weil der generische catch-Block `catch(...)` ***jede Ausnahme abfängt, muss er als letzter in einer Reihe von catch-Blöcken stehen. Jeder*** `catch`***-Block hinter einem generischen*** `catch` ***ist unerreichbar.***

Zusammenfassung

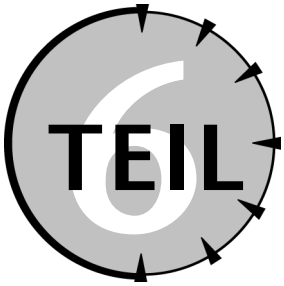
Der Ausnahmemechanismus von C++ stellt einen einfachen, kontrollierten und erweiterbaren Mechanismus zur Fehlerbehandlung dar. Er vermeidet die logische Komplexität, die bei dem Standardmechanismus der Fehlerrückgabewerte entstehen kann. Er stellt außerdem sicher, dass Objekte korrekt vernichtet werden, wenn sie ihre Gültigkeit verlieren.

- Die konventionelle Technik, einen ansonsten ungültigen Wert zurückzugeben, um dem Typ des Fehlers anzuzeigen, hat ernsthafte Begrenzungen. Erstens kann nur eine begrenzte Menge an Information kodiert werden. Zweitens ist die aufrufende Funktion gezwungen, den Fehler zu behandeln, indem er verarbeitet oder zurückgegeben wird, ob sie nun etwas an dem Fehler machen kann oder nicht. Schließlich haben viele Funktionen keinen ungültigen Wert, der so zurückgegeben werden könnte.
- Die Technik der Ausnahmefehler ermöglicht es Funktionen, eine theoretisch nicht begrenzte Anzahl von Informationen zurückzugeben. Wenn eine aufrufende Funktion einen Fehler ignoriert, wird der Fehler die Kette der Funktionsaufrufe nach oben propagiert, bis eine Funktion gefunden wird, die den Fehler behandeln kann.

- Ausnahmen können Unterklassen sein, was die Flexibilität für den Programmierer erhöht.
- Es können mehrere `catch`-Blöcke verkettet werden, um die aufrufende Funktion in die Lage zu versetzen, verschiedene Fehlertypen zu behandeln.

Selbsttest

1. Nennen Sie drei Begrenzungen der Fehlerrückgabetechnik. (Siehe »Konventionelle Fehlerbehandlung«)
2. Nennen Sie die drei Schlüsselwörter, die von der Technik der Ausnahmebehandlung verwendet werden. (Siehe »Wie arbeiten Ausnahmen?«)



Sonntagnachmittag – Zusammenfassung

1. ***Schreiben Sie den Kopierkonstruktor und den Zuweisungsoperator für das Modul*** `Assign-Problem`***. Eine Ressource muss geöffnet werden mit dem*** `nValue`***-Wert des*** `MyClass`***-Objekts, und diese Ressource muss***

   • geöffnet werden bevor sie gültig ist

   • geschlossen werden nachdem sie geöffnet wurde

   • geschlossen werden, bevor sie wieder geöffnet werden kann

   Nehmen Sie an, dass die Prototypfunktionen irgendwo anders definiert sind.

```
// AssignProblem - demonstriert Zuweisungsoperator
#include <stdio.h>
#include <iostream.h>

class MyClass;

// Resource - die Funktion open() bereitet das
//           Objekt auf seinen Gebrauch vor,
//           die Funktion close() »tut es weg«;
//           eine Ressource muss geschlossen werden,
//           bevor sie wieder geöffnet werden kann
class Resource
{
  public:
    Resource();
    void open(int nValue);
    void close();
};

// MyClass - soll vor Gebrauch geöffnet und nach
//          Gebrauch geschlossen werden
class MyClass
{
  public:
    MyClass(int nValue)
    {
        resource.open(nValue);
    }

    ~MyClass()
    {
```

```
        resource.close();
    }

    // Kopierkonstruktor und Zuweisungsoperator
    MyClass(MyClass& mc)
    {
        // ... was kommt hier hin?
    }
    MyClass& operator=(MyClass& s)
    {
        // ...und was hier?
    }

  protected:
    Resource resource;

    // der Wert für resource.open()
    int nValue;
};
```

2. Schreiben Sie einen Inserter für das folgende Programm, das den Nachnamen, den Vornamen und Studenten-ID für die folgende `Student`***-Klasse ausgibt.***

```
// StudentInserter
#include <stdio.h>
#include <iostream.h>
#include <string.h>

// Student
class Student
{
  public:
    Student(char* pszFName, char* pszLName, int nSSNum)
    {
        strncpy(szFName, pszFName, 20);
        strncpy(szLName, pszLName, 20);
        this->nSSNum = nSSNum;
    }

  protected:
    char szLName[20];
    char szFName[20];
    int  nSSNum;
};

int main(int nArgc, char* pszArgs[])
{
    Student student(»Kinsey«, »Lee«, 1234);
    cout << »Mein Freund ist » << student << »\n«;
    return 0;
}
```