IV.

Teil IV: Das C++-Rezeptbuch: Tipps und Lösungen für typische Aufgaben

20

Sichere Programmentwicklung

Dieses Kapitel behandelt die folgenden Themen:

- Regeln zum Design von Methoden
- Defensive Programmierung
- Exception-sichere Ressourcenbeschaffung
- Tipps zur Thread-Prgrammierung

Dieses Kapitel zeigt eine Auswahl von Tipps zu sicheren Programmentwicklung. Darunter sind einfache Tipps wie der zum korrekten Vergleich von `double`-Werten oder zur Schreibweise von `if`-Anweisungen, aber auch Faustregeln zur Konstruktion von Methoden und zur exception-sicheren Ressourcenbeschaffung.

20.1 Regeln zum Design von Methoden

Dieser Abschnitt gibt einige Empfehlungen zur Konstruktion der Prototypen von Methoden. Zur Erinnerung: Ein Prototyp ist die Deklaration einer Schnittstelle, über die von außen auf ein Objekt zugegriffen werden kann. Ein Prototyp besteht aus Rückgabe-

typ, Name und Parameterliste, wie schon aus Abschnitt 3.1.1 bekannt ist. Dazu kommt manchmal noch ein `const`-Qualifizierer. Unter anderem beziehen sich die Empfehlungen auf diese vier Bestandteile. Als Beispiel in einigen Fällen dient die Klasse `Geld`, deren Objekte aus einem Geldbetrag und der Währung als Attribute bestehen. Die Klasse soll in einer Anwendung Folgendes leisten:

```
// main()-Programm
Geld kaufpreis(100.00, "Euro");
cout << kaufpreis.betrag();   // 100
cout << kaufpreis.waehrung(); // Euro
// Ausgabe für Scheckvordrucke:
cout << kaufpreis.toString(); // eins-null-null
kaufpreis.neuerbetrag(90.0);  // Betrag ändern
```

Aus der Anwendung ergeben sich Prototypen, die im `public`-Bereich der Klasse aufgeführt sind. Die Namen der privaten Attribute sind beliebig gewählt.

```
class Geld {
   public:
     Geld(double einBetrag, const string& eineWaehrung);
     double betrag() const;
     void neuerBetrag(double);
     const string& waehrung() const;
     string toString() const;
   private:
     double derBetrag;
     string dieWaehrung;
};
```

Die Methode `toString()` berechnet aus dem Betrag einen String entsprechend der Ziffernfolge des ganzzahligen Anteils des Betrags und gibt ihn zurück. Hier nun die sechs Regeln:

1. Die geplante Anwendung bestimmt die Methoden. Beim Entwurf einer Klasse ist es empfehlenswert, sich die Anwendung vorher zu überlegen und nur solche Prototypen vorzusehen, die dazu beitragen. Es ist nicht sinnvoll, Methodendeklarationen aufzuschreiben, die nur *vielleicht* brauchbar sind und deren Anwendung ungewiss ist. Der Grund dafür ist einfach: Alle Methoden müssen irgendwann auch definiert, dokumentiert und getestet werden. Diese Arbeit kann man sich für alle Methoden sparen, die nicht zum Einsatz kommen.
2. Der Name einer Methode soll beschreiben, was die Methode tut. Zum Beispiel ist der Name `betrag` aussagekräftiger als etwa `zahl`.
3. Eine Methode, die den Zustand eines Objekts nicht ändert, erhält den `const`-Qualifizierer (Methoden `betrag()`, `waehrung()` und `toString()`, nicht aber `neuerBetrag()`). Siehe auch die ausführliche Diskussion in Abschnitt 4.5.
4. Der Rückgabetyp ist natürlich `void`, wenn die Methode zwar etwas tut, aber nichts zurückgibt. Andernfalls bestimmt sich der Rückgabetyp aus der Antwort auf die Frage: Ist der zurückgegebene Wert ein Attribut der Klasse?
   - **Ja:** Nächste Frage: Ist der zurückgegebene Wert von einem Grunddatentyp? (d.h. `int`, `double`, `bool` usw. im Gegensatz zu einem Klassentyp)
     - *Ja:* Rückgabe per Wert. Beispiel: `betrag()`

- *Nein:* Rückgabe per `const&`. Beispiel: `waehrung()`
- **Nein** (kein Attribut der Klasse): Rückgabe per Wert. Beispiel: `toString()`

5. Die Art der Übergabe eines Objekts in der Parameterliste kann ebenfalls durch die Beantwortung einiger Fragen bestimmt werden. Soll das Objekt beim Aufrufer der Methode verändert werden?
   - **Ja:** Übergabe per Referenz (siehe Seite 111)
   - **Nein:** Nächste Frage: Gehört der übergebene Wert zu einem Grunddatentyp?
     - *Ja:* Übergabe per Wert. Beispiel: `neuerBetrag(double)`
     - *Nein:* Wird in der Funktion eine lokale Kopie des Parameters benötigt?
       - *Ja:* Übergabe per Wert (vgl. Diskussionen auf den Seiten 168 und 590).
       - *Nein:* Übergabe per `const&`, siehe Übergabe des Strings im Konstruktor.
6. Wenn Operationen verkettet werden sollen, ist das Objekt selbst per Referenz zurückzugeben. In C++ sind Anweisungen wie `a = b = c;` erlaubt und üblich, ebenso wie die schon bekannte Verkettung von Ausgaben mehrerer Variablen, etwa `cout << a << b << c << endl;`. Die Verkettung von Operatoren wird ausführlich in Kapitel 9 diskutiert. Operatoren sind Funktionen, für die dasselbe gilt. Nehmen wir zum Beispiel die Anwendung der Methode

```
// Methode der Klasse Rational
void add(const Rational& r);
```

von Seite 164 auf rationale Zahlen `a`, `b` und `c`. Die Operation `a.add(b);` ist damit möglich, nicht aber Verkettungen wie zum Beispiel

```
a.add(b).add(c); // a) entspricht: a += b; a += c;
a.add(b.add(c)); // b) entspricht: a += b += c;
```

Die Verkettungen könnten natürlich durch jeweils zwei einzelne Anweisungen ersetzt werden, aber hier geht es darum, zu zeigen, dass und wie Verkettungen mit der Rückgabe des Objekts selbst möglich gemacht werden. Betrachten wir beide Fälle:

(a) Diese Anweisung wird von links abgearbeitet. `a.add(b)` muss dann ein Objekt zurückliefern, auf das dann `add(c)` angewendet wird. Auf einen Rückgabetyp `void` lässt sich keine Funktion anwenden, und der Compiler würde sich bitter beschweren. Dieses zurückgegebene Objekt ist sinnvollerweise nichts anderes als das durch die Addition veränderte Objekt `a`, hier `a` genannt. Aus `a.add(b)` ergibt sich also `a`, für das `add(c)` aufgerufen wird. In Einzelschritte zerlegt:

```
a.add(b).add(c);
  a.add(c);
```

Damit ergib sich eine veränderte Implementation (und Deklaration) für `add()`:

```
// veränderter Rückgabetyp:
Rational& Rational::add(const Rational& r) {
    zaehler = zaehler*r.nenner + r.zaehler*nenner;
    nenner = nenner*r.nenner;
    kuerzen();
    return *this;
}
```

Mit `*this` wird das Objekt, für das die Methode aufgerufen wird, zurückgegeben.

(b) Ein Aufruf der Form `a.add(x)` verlangt, dass das Argument `x` vom Typ `Rational` ist. Da `x` identisch mit `b.add(c)` und gemeint ist, dass erst `c` auf `b` addiert wird, welches dann auf `a` addiert wird, folgt daraus, dass der Aufruf `b.add(c)` das veränderte `b` zurückgeben muss. Die unter a) angegebene Implementierung löst auch dieses Problem.

Natürlich kann von den Empfehlungen abgewichen werden – wenn es einen Grund dafür gibt. *Kleine* Klassenobjekte kosten nicht viel Zeit bei der Kopie und können daher per Wert übergeben werden statt per Referenz auf `const`.

20.2 Defensive Programmierung

Defensive Programmierung soll die Risiken falscher Algorithmen oder einer falschen Benutzung vermindern. So kann der »Buffer Overflow«, ein Fehler, der vielen Hackern Angriffsmöglichkeiten geboten hat und vermutlich noch bietet, durch defensive Programmierung verhindert werden. Einen Buffer Overflow zeigen die Beispiele 1 bis 3 auf Seite 194 ff. zur Eingabe einer Zeichenkette; Beispiel 4 zeigt, wie er verhindert wird. In diesem Abschnitt wird eine Auswahl weiterer Techniken genannt.

Allgemeine Empfehlungen

Programmcode kann schon durch vielerlei Maßnahmen verbessert werden. Manche davon können sehr aufwendig sein, wie etwa das Testen von Software, manche sind einfacherer Art. Eine kleine Auswahl:

- Programmcode verständlich und gut lesbar schreiben.
- Nicht das Rad neu erfinden, sondern erprobte Bibliotheken verwenden.
- Vier Augen sehen mehr als zwei. Ein Review des Codes durch eine *andere* Person als den Programmautor kann viele Fehler oder kritische Stellen finden.
- Eine damit verwandte Möglichkeit ist das »Pair Programming«, bei dem einer den Code schreibt und der andere über das zu lösende Problem nachdenkt, das Geschriebene kontrolliert und alles, was ihm dabei auffällt, gleich mit dem Autor bespricht. Beide sollen sich abwechseln. Pair Programming ist besonders durch das »Extreme Programming«-Buch von Kent Beck [Beck] bekannt geworden.
- Vor- und Nachbedingungen und Invarianten prüfen, sofern vertretbar (zur Diskussion des Aufwands siehe Seite 315).
- Exception-sicher programmieren (siehe Seiten 315 und 567).
- Nicht-reparierbare Fehler mit einer Exception melden (nicht mit Assertions).

Um nicht im Allgemeinen zu bleiben, gibt es im Folgenden eine Auswahl ganz konkreter Tipps, meistens ergänzt durch Programmbeispiele.

20.2.1 double- und float-Werte richtig vergleichen

Auf Seite 75 ist zu sehen, wie ein schlechter Vergleich zu undefiniertem Verhalten einer Schleife führen kann. Die dort gezogene Schlussfolgerung ist »Verwenden Sie nur integrale Datentypen wie `int`, `size_t` oder `char` als Laufvariable in Schleifen!«. Nur integrale Datentypen zu verwenden, hilft nicht, wenn tatsächlich `double`- oder `float`-Werte verglichen werden müssen, um in Abhängigkeit vom Vergleich Entscheidungen zu treffen. Dabei sind die Vergleiche `<`, `<=`, `>` und `>=` sicher. Problematisch sind `==` und `!=`, weil nur Bitmuster verglichen werden, ungeachtet der begrenzten Genauigkeit von Rechnungen und der Zahlendarstellung. Empfehlung: Vor der Rechnung den gewünschten Bereich festlegen, innerhalb dessen zwei Werte als gleich betrachtet werden sollen. Mathematiker verwenden oft »Epsilon« als Namen für einen solchen Bereich:

```
// x und y seien die zu vergleichenden Werte
double eps = 1.0e-8;     // Bereich
if(fabs(x-y) < eps) {   // fabs : Absolutbetrag
   // x und y werden als gleich betrachtet
}
else {
   // x und y sind ungleich
}
```

Ähnlich kann man vorgehen, wenn es um eine *relative* Abweichung gehen soll, zum Beispiel soll `y` nicht mehr als 0,1 Prozent von `x` abweichen:

```
double eps = 0.001;          // 0,1 %
if(fabs((y-x)/x) < eps) {  // Risiko, siehe Text
   // x und y werden als gleich betrachtet
}
```

Der Bezugswert `x` darf nicht zu klein werden, um einen Divisions-Overflow zu vermeiden (bzw. einen Underflow bei Multiplikation beider Seiten mit `x`).

20.2.2 const verwenden

Um versehentliche Änderungen zu vermeiden, sollten alle unveränderliche Variablen und alle Methoden, die ein Objekt nicht verändern, `const` sein. Vorteil: Schon der Compiler prüft die Einhaltung der Absicht.

20.2.3 Anweisungen nach for/if/while einklammern

Nachträgliche Ergänzungen können ohne die Klammern leicht zu einem Fehler führen. Angenommen, dass in der Anweisung

```
if(x == 1)
   a = a + 3;
```

auch noch die Variable `counter` hochgezählt werden soll. Dies hat sich erst viel später herausgestellt, sodass nachträglich die `if`-Anweisung ergänzt wird:

```
if(x == 1)
   a = a + 3;
   ++counter;                    // falsch
```

Tja, mal eben schnell ein Programm »verbessert«! Der Effekt: `++counter;` wird stets ausgeführt, egal ob `x` gleich 1 ist oder nicht! Hier fehlen die geschweiften Klammern. Daher gibt es die folgende *Empfehlung zur Schreibweise*. `if`-Anweisungen sollen stets mit geschweiften Klammern geschrieben werden, also zum Beispiel

```
if(x == 1) {
   a = a + 3;
}
```

Dies gilt auch, wenn der Inhalt des Blocks aus nur einer Zeile besteht. Einige der oben erwähnten Fehler entstehen dadurch gar nicht erst[1].

20.2.4 int und unsigned/size_t nicht mischen

Für die Anzahl der Objekte in einem Container wird im Allgemeinen der Typ `size_t` verwendet, weil die Anzahl nie negativ sein kann. Wenn so ein Wert unüberlegt mit einem `int`-Wert verglichen wird, kann es schwer zu entdeckende Fehler geben. Ein Beispiel finden Sie auf Seite 66.

20.2.5 size_t oder auto statt unsigned int verwenden

`size_t` ist groß genug, um die Größe eines jeden Objekts des aktuellen Systems aufzunehmen. Das ist bei `unsigned int` nicht garantiert. Das wirkt sich auch auf Konstante aus, zum Beispiel auf `string::npos`, das als Kriterium für eine erfolgreiche Suche verwendet wird. `npos` ist typischerweise die maximal mögliche `unsigned`-Zahl. Beispiel:

```
string zudurchsuchenderText("www.cppbuch.de");
unsigned int punktPosition = zudurchsuchenderText.find("."); // ??
if(punktPosition == string::npos) {
   cout << "'.' nicht gefunden!" << endl;
}
```

Das mag auf einem 32-Bit-System funktionieren, auf dem der Typ von `npos` derselbe Typ wie `unsigned int` ist. Auf einem 64-Bit-System jedoch könnte `unsigned int` eine 32-Bit-Zahl sein und `npos` eine 64-Bit-Zahl. Das heißt, auf so einem 64-Bit-System wäre die Bedingung *niemals* wahr, ob der zu durchsuchende Text einen Punkt enthält oder nicht.

Der tatsächliche Typ von `npos` ist `string::size_type`. `size_t` ist groß genug, und wenn der Typ von `size_t` nicht derselbe wie `string::size_type` sein sollte, gibt es auf jeden Fall eine definierte Typumwandlung. Noch besser ist es aber, den Compiler mit `auto` den richtigen Typ feststellen zu lassen:

```
auto punktPosition = zudurchsuchenderText.find("."); // !!
```

20.2.6 Postfix++ mit Präfix++ implementieren

`++i` und `i++` unterscheiden sich nur darin, dass im ersten Fall `i` erst hochgezählt und dann verwendet wird. Im zweiten Fall ist es umgekehrt. Um genau dieselbe Bedeutung des »Hochzählens«, das in einem binären Baum das Weiterschalten zum nächsten Schlüssel und bei einem Datum den nächsten Tag bedeuten kann, ohne Code-Replikation zu

[1] In diesem Buch wird aus Platzgründen gelegentlich davon abgewichen.

erreichen, ist die auf Seite 336 angegebene Methode gut geeignet. Der nachgestellte Operator `operator++(int)` wird mithilfe des Präfix-Operators implementiert:

```
X X::operator++(int) { // X hochzählen, Parameter wird nicht gebraucht
   X temp(*this);     // temporäres Objekt erzeugen
   ++*this;           // Präfix ++ aufrufen, *this wird modifiziert
   return temp;       // vorherigen Zustand zurückgeben
}
```

`X` kann jede Klasse sein, die einen Kopierkonstruktor und `operator++()` besitzt. Weil ein temporäres Objekt erzeugt wird, sollte man den nachgestellten `++`-Operator nicht als Ersatz für den vorangestellten `++`-Operator verwenden. Beispiel:

```
for(Iterator i = container.begin();
            i != container.end(); i++) { // ungünstig
   // ...
}
for(Iterator i = container.begin();
            i != container.end(); ++i) { // besser
   // ...
}
```

Der Compiler kann die Optimierung bei den Grunddatentypen selbst vornehmen; bei komplexeren Strukturen kann er das nicht ohne Weiteres: Ohne eine aufwendige Datenflussanalyse ist nicht feststellbar, ob die Bedeutung des Hochzählens in beiden Fällen exakt dieselbe ist.

20.2.7 Ein Destruktor darf keine Exception werfen

Die Einhaltung dieser Empfehlung garantiert, dass Destruktoren exception-sicher sind. Zur Erläuterung sei als Beispiel eine Klasse `Unsicher` mit zwei dynamischen Variablen gegeben:

```
class Unsicher {
public:
   Unsicher(int c, int d)
   : a(new X(c)), b(new X(d)) {
   }
   ~Unsicher() {
      delete a;       // siehe Text
      delete b;
   }
   // weggelassen: Methoden, die die Attribute ändern/verwenden
private:
   X* a;
   X* b;
};
```

`delete a;` ruft den Destruktor der Klasse `X` auf. Wenn dieser eine Exception wirft, wirft auch `~Unsicher()` eine Exception, und `delete b;` wird nie ausgeführt – ein Speicherleck. Eine `try/catch`-Konstruktion der Art

```
   ~Unsicher() {
      try {          // hilft nicht
```

```
        delete a;
        delete b;
    } catch(...) {
        ; // nichts tun
    }
}
```

würde nicht helfen: Zwar könnte die Exception den Destruktor der Klasse `Unsicher` nicht verlassen, aber `delete b;` wird auch jetzt nicht ausgeführt, wenn `delete a;` eine Exception wirft. Also: Grundsätzlich jeden Destruktor so schreiben, dass er keine Exception wirft.

20.2.8 Typumwandlungsoperatoren vermeiden

Um die Typprüfung des Compilers nicht zu umgehen, sollte man Typumwandlungsoperatoren nur in begründeten Fällen einsetzen. Jede vom Compiler nicht gefundene Typverletzung ist eine verlorene Chance, einen möglichen Fehler zu finden. Wenn eine Typumwandlung gewünscht ist, ist es am besten, auf den Typumwandlungsoperator zu verzichten und stattdessen eine eigene Methode dafür zu schreiben. Eine implizite Typumwandlung ist dann unmöglich (siehe auch Übungsaufgabe von Seite 339).

20.2.9 explicit-Konstruktoren bevorzugen

Die Argumente des vorhergehenden Abschnitt gelten auch hier. Ein weiterer Vorteil ist die bessere Lesbarkeit des Programmcodes, weil sofort gesehen wird, dass eine Typumwandlung beabsichtigt ist. Ein Beispiel sehen Sie auf Seite 161.

20.2.10 Leere Standardkonstruktoren vermeiden

Objekte werden vom Konstruktor initialisiert. Manche IDEs schlagen gleich einen parameterlosen (Standard-)Konstruktor vor, oder er wird gedankenlos hingeschrieben (kann ja nicht schaden ...). Wenn er keinen Inhalt hat *und* ein weiterer Konstruktor existiert, können Flüchtigkeitsfehler produziert werden. Ein Beispiel: Ein Punkt soll stets definierte Koordinaten besitzen. Falls ein zusätzlicher leerer Standardkonstruktor existiert, kann die folgende Situation eintreten, wenn die Initialisierungsdaten vergessen werden:

```
Punkt p1(100, 200);      // ok, definierte Koordinaten
Punkt p2;                // undefinierte Koordinaten, nicht beabsichtigt
```

Bei Nichtvorhandensein des leeren Standardkonstruktors würde der Compiler bei `p2` eine Fehlermeldung erzeugen.

20.2.11 Kopieren und Zuweisung verbieten

Klassen mit einem Zeiger als Attribut verwalten dynamisch angelegte Daten. Wie auf Seite 236 erläutert, benötigen diese in der Regel außer einem Destruktor einen eigenen Kopierkonstruktor und Zuweisungsoperator. Wenn versehentlich auf diese Elemente verzichtet wird, werden sie automatisch vom System erzeugt. Das kann katastrophale Folgen haben, weil automatisch erzeugte Kopierkonstruktoren und Zuweisungsoperatoren nur den Zeiger kopieren, nicht aber, auf was er zeigt. Die linke Seite der Abbildung 6.2 auf Seite 236 zeigt die Wirkung. Die folgende Klasse verhält sich wie in der Abbildung gezeigt:

```
class intArray { // fehlerhaft, siehe Text
 public:
   intArray(int* p, int anz)
      : anzahl(anz), iptr(p) {
   }
   ~intArray() { delete[] iptr;}
   int& operator[](int i) { return *(iptr + i);}
 private:
   size_t anzahl;
   int* iptr;
};

// mögliche Anwendung:
int main() {
   intArray iArr1(new int[10], 10);
   iArr1[0] = 100;
}
```

Wenn nun eine Kopie erzeugt wird, indem zu `main()` die Zeile

```
   intArray iArr2 = iArr1;
```

hinzugefügt wird, ist das Programm fehlerhaft. Eine nachfolgende Änderung von `iArr1` wirkt sich bei `iArr2` aus und umgekehrt. Außerdem gibt es einen Crash am Ende des Programms, weil der Destruktor zweimal dasselbe Array zu löschen versucht.

Wenn nun gar keine Kopien gebraucht werden, müsste man also einen eigenen Kopierkonstruktor und Zuweisungsoperator schreiben, nur um deren automatische Generierung zu verhindern. Es gibt aber zwei bessere Lösungen. Die erste ist, Kopierkonstruktor und Zuweisungsoperator privat zu deklarieren; eine Implementierung ist nicht erforderlich:

```
class intArray {
 public:
   // wie oben ...
 private:
   intArray(intArray&);      // verbietet Kopie
   void operator=(intArray&); // verbietet Zuweisung
   size_t anzahl;
   int* iptr;
};
```

Alternativ wird dem Compiler mit `= delete` mitgeteilt, dass er auf die automatische Erzeugung verzichten soll:

```
class intArray {
 public:
   intArray(intArray&) = delete;
   void operator=(intArray&) = delete;
   intArray(int* p, int anz)
      : anzahl(anz), iptr(p) {
   }
   ~intArray() { delete[] iptr;}
   int& operator[](int i) { return *(iptr + i);}
 private:
```

```
   size_t anzahl;
   int* iptr;
};
```

20.2.12 Vererbung verbieten

Manchmal möchte man die Ableitung von einer Klasse verhindern. Wenn zum Beispiel eine Klasse keinen virtuellen Destruktor hat, sind von ihr abgeleitete Klassen nicht polymorph ohne die Gefahr von Speicherlecks nutzbar, wie in Abschnitt 7.6.3 belegt, sodass man eine versehentliche polymorphe Nutzung ausschließen möchte. Ein anderer Grund mag sein, dass die Semantik von Funktionen nicht durch Redefinition in einer abgeleiteten Klasse verändert werden soll. Man kann nach [Str94] die Tatsache ausnutzen, dass bei virtuellen Basisklassen der Konstruktor der am meisten abgeleiteten Klasse für die Konstruktion des Basisklassensubobjekts verantwortlich ist (siehe Seite 291). Beispiel:

```
class Final {
   friend class NichtAbleitbar;
private:
   Final() {}
};

class NichtAbleitbar : public virtual Final {
public:
   NichtAbleitbar(int i): attr(i) {}
   int get() const { return attr; }
private:
   int attr;
};

class Versuch : public NichtAbleitbar {
public:
   Versuch(int i)
   : NichtAbleitbar(i) { // Fehlermeldung des Compilers
   }
};
```

Weil die Klasse `NichtAbleitbar` von `Final` erbt, muss sie auf deren Konstruktor zugreifen können. Da dieser privat ist, wird der Zugriff über die `friend`-Deklaration ermöglicht. Die Klasse `Versuch` ist zwar für die Konstruktion des Basisklassensubobjekts verantwortlich, hat aber keinen Zugriff – daher die Fehlermeldung des Compilers. Die virtuelle Vererbung ist entscheidend.

20.2.13 Defensiv Objekte löschen

Hier geht es um das Problem, dass ein Zeiger nach Löschen des Objekts weiterhin zugreifbar bleibt:

```
int *pa = new int[4];    // Array von int-Zahlen
// ... verwenden
delete [] pa;  // löschen
// .. mehr Programmcode
if(pa == 0) {            // Fehler
```

```
...
}
```

Der Fehler liegt in dem undefinierten Wert von `pa` nach der Löschoperation. Falls ein Zeiger nach dem Löschen noch verwendet werden kann, sollte man ihn direkt nach dem `delete` mit `pa = 0;` auf NULL setzen. Dann kann er auf 0 geprüft werden oder es gibt eine definierte Fehlermeldung.

20.3 Exception-sichere Beschaffung von Ressourcen

Wenn eine Ressource beschafft werden soll, kann ein Problem auftreten. Das kann eine Datei sein, die nicht gefunden wird, oder ein Fehlschlag beim Beschaffen von Speicher. Weil die Probleme strukturell ähnlich sind, beschränke ich mich hier auf Probleme bei der dynamischen Beschaffung von Speicher. Das kann in einer Methode oder auch schon im Konstruktor auftreten. Ziel ist es, beim Auftreten von Exceptions kein Speicherleck zu erzeugen und die betroffenen Objekte in ihrem Zustand zu belassen.

20.3.1 Sichere Verwendung von shared_ptr

Bei der Konstruktion eines `shared_ptr` (Beschreibung Seite 851) soll die Erzeugung des Zeigers mit `new` stets innerhalb der Parameterliste geschehen.

```
Ressource pr = new Ressource(id);
// ... weiterer Code
shared_ptr<Ressource> sptr(pr);          // 1. falsch!
```

```
shared_ptr<Ressource> p(new Ressource(id)); // 2. richtig!
```

Begründung: Im Fall 1 kann es die folgenden Fehler geben:

- Es wäre möglich, `delete pr` aufzurufen. Bei der Zerstörung von `sptr` wird der Destruktor für `*pr` auch aufgerufen, dann also insgesamt *zweimal*.
- Es könnte sein, dass im Bereich »weiterer Code« eine Exception auftritt. Der resultierende Sprung des Programmablaufs aus dem aktuellen Kontext führt dazu, dass `delete` nicht mehr möglich ist. Das erzeugte Objekt bleibt unerreichbar im Speicher.

Im Fall 2 kann dies nicht geschehen: Wenn eine Exception geworfen wird, werden automatisch die Destruktoren aller auf dem Laufzeit-Stack befindlichen Objekte des verlassenen Gültigkeitsbereichs aufgerufen, also auch der Destruktor des `shared_ptr`-Objekts, der wiederum für das Löschen des übergebenen Objekts sorgt – eine Realisierung des Prinzips »Resource Acquisition Is Initialization« (RAII, siehe Glossar).

20.3.2 shared_ptr für Arrays korrekt verwenden

Der Destruktor eines `shared_ptr`-Objekts wendet `delete` auf den intern gespeicherten Zeiger an, wenn kein anderer `shared_ptr` auf die Ressource verweist (siehe Seite 851). Dies

kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 203 beschrieben. Hier ein Beispiel:

```
int* p = new int[10];
// p verwenden
// delete p; falsch!
delete [] p;      // richtig
```

Zwar kann es sein, dass im Fall der falschen Anweisung das Speicherleck nicht bemerkt wird, oder dass der Compiler aus dem Kontext den Fehler erkennt und korrigiert. Nach [ISOC++] ist das Verhalten jedoch undefiniert, das heißt, alle Möglichkeiten vom Weiterlaufen des Programms bis zum Absturz des Programms sind »legal«. Damit ist auch das Verhalten des folgenden Programms undefiniert:

```
void funktion() {
   shared_ptr<int> p(new int[10]);                    // falsch
   // ... etliche Zeilen weggelassen
}   // Memory-Leak möglich
```

Die Lösung des Problems besteht in der Übergabe eines `deleter`-Objekts an den `shared_ptr`. Wenn es so ein Funktionsobjekt gibt, wird dessen `operator()()` aufgerufen.

```
template<typename T>
struct ArrayDeleter {
   void operator()(T* ptr) {
      delete [] ptr;
   }
};

void funktion() {
   shared_ptr<int> p(new int[10], ArrayDeleter<int>()); // richtig
   // ... etliche Zeilen weggelassen
}  // kein Memory-Leak, Array wird korrekt gelöscht
```

20.3.3 unique_ptr für Arrays korrekt verwenden

Bei `unique_ptr`-Objekten tritt dieselbe Problematik auf, wie oben im Abschnitt 20.3.1 beschrieben. Die Schnittstelle ist etwas anders:

```
template <class T, class D = default_delete<T>> class unique_ptr;
```

Der Typ des für die Löschung zuständigen Objekts gehört zur Schnittstelle. Wenn ein Arraytyp, gekennzeichnet durch `[]`, eingesetzt wird, kann der zweite Typ entfallen. Er wird dann durch den vorgegebenen (default) Typ für den Deleter ersetzt, der `delete []` aufruft. Die Funktion `f()` zeigt, wie es geht.

Listing 20.1: unique_ptr und Array

```
// cppbuch/k33/uniqueptr/array.cpp
#include<memory>

void f() {
   std::unique_ptr<int[]> arr(new int[10]); // int[] statt int
   // Benutzung des Arrays
```

```
   for(int i = 0; i < 10; ++i) {
      arr[i] = i;
      std::cout << arr[i] << std::endl;
   }
}  // kein Memory-Leak, Array wird korrekt gelöscht

int main() {
   f();
}
```

Um den default-Template-Parameter sichtbar zu machen, könnte die erste Zeile in `f()` so geschrieben werden:

```
   std::unique_ptr<int[], std::default_delete<int[]>> arr(new int[10]);
```

20.3.4 Exception-sichere Funktion

```
void func() {              // fehlerhaft, siehe Text
     Datum heute;                 // Stack-Objekt
     Datum *pD = new Datum;       // Heap-Objekt beschaffen
     heute.aktuell();             // irgendeine Berechnung
     pD->aktuell();               // irgendeine Berechnung
     delete pD;                   // Heap-Objekt freigeben
}
```

Wenn die Funktion `aktuell()` eine Ausnahme auswirft, wird der Destruktor von Objekt `heute` gerufen, und das Objekt wird vom Stack geräumt. Das Objekt, auf das `pD` zeigt, wird jedoch niemals freigegeben, weil `delete` nicht mehr erreicht wird und `pD` außerhalb des Blocks unbekannt ist:

```
int main() {
    try {
       func();
    }
    catch(...) {
       //... pD ist hier unbekannt
    }
}
```

Aus diesem Grund sollten ausschließlich Stack-Objekte (automatische Objekte) verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn man Beschaffung und Freigabe eines dynamischen Objekts innerhalb eines Stack-Objekts versteckt. Das Hilfsmittel dazu kennen wir bereits, nämlich die »intelligenten« Zeiger aus Abschnitt 9.5:

```
void func() {
     // shared_ptr der Standardbibliothek, siehe Abschnitt 9.5.1. Header: <memory>
     std::shared_ptr<Datum> pDshared(new Datum);
     pDshared->aktuell();           // irgendeine Berechnung
}
```

Nun ist `pDshared` ein automatisches Objekt. Wenn jetzt eine Exception auftritt, gibt es kein Speicherleck, weil der Destruktor von `pDshared` den beschafften Speicher freigibt.

20.3.5 Exception-sicherer Konstruktor

Das Ziel, den Zustand eines Objekts bei Auftreten einer Exception unverändert zu lassen, ist in diesem Fall nicht erreichbar – das Objekt wird ja erst durch den Konstruktor erzeugt. Es geht also darum, dass

1. Ressourcen, die innerhalb des Konstruktors beschafft werden, freigegeben werden, und dass
2. Exceptions beim Aufrufer aufgefangen werden können.

In diesem Zusammenhang ist es wichtig zu wissen, wie sich C++ verhält, wenn in einem Konstruktor eine Exception auftritt.

Verhalten bei einer Exception im Konstruktor

- Für alle vollständig erzeugten (Sub-)Objekte wird der Destruktor aufgerufen. Unter »vollständig erzeugt« ist zu verstehen, dass der Konstruktor bis zum Ende durchlaufen wurde.
- Für *nicht* vollständig erzeugte (Sub-)Objekte wird *kein* Destruktor aufgerufen.

Das folgende Beispiel demonstriert dieses Verhalten. Ein Objekt der Klasse `Ganzes` enthält zwei Subobjekte der Typen `Teil1` und `Teil2`, die wie folgt definiert sind. Beachten Sie, dass der Konstruktor von `Teil2` zur Demonstration eine Exception wirft!

Listing 20.2: Klassen Teil1 und Teil2

```
// cppbuch/k20/teil.h
#ifndef TEIL_H
#define TEIL_H
#include<iostream>

class Teil1 {
public:
   Teil1(int x)
      : attr(x) {
   }
   ~Teil1() {
      std::cout << "Teil1::Destruktor gerufen!" << std::endl;
   }
private:
   int attr;
};

class Teil2 {    // Konstruktor wirft zur Demonstration eine Exception
public:
   Teil2() {
      throw std::exception(); // auskommentieren:
      // dann wird der Destruktor gerufen, ansonsten NICHT!
   }
   ~Teil2() {
      std::cout << "Teil2::Destruktor gerufen!" << std::endl;
   }
};
#endif
```

Bei der Konstruktion des Objektes `ganzes` (siehe Beispiel unten) wird das Subobjekt `teil1` initialisiert. Die Konstruktion des Subobjekts vom Typ `Teil2`, die mit `new` versucht wird, schlägt jedoch fehl, weil der `Teil2`-Konstruktor eine Exception wirft.

Listing 20.3: Exception im Konstruktor

```
// cppbuch/k20/excImKonstruktor1.cpp
#include<iostream>
#include"teil.h"
using namespace std;

class Ganzes {
public:
   Ganzes() : teil1(99) {
      ptr = new Teil2;
   }
   ~Ganzes() {
      cout << "Ganzes::Destruktor gerufen!" << endl;
      delete ptr;
   }
private:
   Teil1 teil1;
   Teil2* ptr;
   Ganzes(const Ganzes&);          // für Beispiel nicht erforderlich
   Ganzes& operator=(const Ganzes&); // für Beispiel nicht erforderlich
};

int main() {
   try {
      Ganzes ganzes;
   }
   catch(const exception& e) {
      cout << "Exception gefangen: " << e.what() << endl;
   }
}
```

Weil nur das Subobjekt `teil1` vollständig konstruiert wird, kommt auch nur dessen Destruktor zum Tragen. Die anderen Destruktoren werden nicht aufgerufen. Wird nun die Anweisung `throw exception();` auskommentiert oder gelöscht, werden alle Destruktoren aufgerufen.

Die am Anfang dieses Abschnitts genannten Punkte werden in diesem Beispiel bei Auftreten der Exception erreicht: Die »Ressource« `teil1` wird freigegeben, und die Exception wird in `main()` aufgefangen.

Ein Gegenbeispiel: Wenn die Klasse `Ganzes` *beide* Sub-Objekte mit `new` erzeugen würde, aber so, dass erst die Erzeugung des zweiten Sub-Objekts eine Exception wirft, wird der Destruktor für das erste Sub-Objekt *nicht* aufgerufen. Der Speicherplatz wird nicht wieder freigegeben.

Listing 20.4: Fehlerhafter Konstruktor

```
// Auszug aus cppbuch/k20/excImKonstruktor2.cpp
class GanzesMitFehler {
public:
   GanzesMitFehler() : ptr1(new Teil1(99)),
                       ptr2(new Teil2) {
   }
   ~GanzesMitFehler() {
      cout << "GanzesMitFehler::Destruktor gerufen!" << endl;
      delete ptr1;
      delete ptr2;
   }
private:
   Teil1* ptr1;
   Teil2* ptr2;
   // Kopierkonstruktor und Zuweisungsoperator weggelassen
};
```

Mit `shared_ptr` wie in Abschnitt 20.3.1 geht man sicher. Wenn der `Teil2`-Konstruktor eine Exception wirft, wird der Destruktor von `Teil1` aufgerufen und gibt den Speicherplatz frei:

Listing 20.5: Konstruktor mit shared_ptr

```
// Auszug aus cppbuch/k20/excImKonstruktor3.cpp
class GanzesKorrigiert {
public:
   GanzesKorrigiert() : ptr1(new Teil1(99)),
                        ptr2(new Teil2) {
   }
   ~GanzesKorrigiert() {
      cout << "GanzesKorrigiert::Destruktor gerufen!" << endl;
       // delete nicht notwendig wegen shared_ptr
   }
private:
   shared_ptr<Teil1> ptr1;
   shared_ptr<Teil2> ptr2;
};
```

20.3.6 Exception-sichere Zuweisung

Wenn bei einer Kopie Speicher beschafft werden muss, sollte man das zuerst tun! Der Grund: Falls es dabei eine Exception geben sollte, würden alle nachfolgenden, den Zustand des Objekts verändernden Anweisungen gar nicht erst ausgeführt. Die Problematik findet sich typischerweise beim Kopierkonstruktor und dem Zuweisungsoperator. Dazu gehören auch der Kurzformoperator `+=` und die Bildung temporärer Objekte. Sehen wir uns dazu eine mögliche Lösung der Aufgabe 9.6 von Seite 331 an:

```
// exception-sicher
MeinString& MeinString::operator=(const MeinString& m) { // Zuweisung
    char *p = new char[m.len+1]; // zuerst neuen Platz beschaffen
    strcpy(p, m.start);          // kopieren
```

```
    delete [] start;           // alten Platz freigeben
    len   = m.len;             // Verwaltungsinformation aktualisieren
    start = p;
    return *this;
}
```

Man könnte vordergründig daran denken, erst den alten Platz freizugeben, weil er ohnehin nicht mehr gebraucht wird, und dabei in der Summe sogar Speicher sparen, wenn nämlich bei `new` der alte Speicherplatz wiederverwendet werden sollte. Auch bräuchte man die Variable `p` nicht:

```
// NICHT exception-sicher!
MeinString& MeinString::operator=(const MeinString& m) { // Zuweisung
    delete [] start;           // weg damit, es wird schon gutgehen!
    start = new char[m.len+1]; // neuen Platz beschaffen
    strcpy(start, m.start);
    len   = m.len;             // Verwaltungsinformation aktualisieren
    return *this;
}
```

Wenn bei der Speicherplatzbeschaffung ein Problem auftreten sollte, wäre der Inhalt des Objekts durch das direkt vorangegangene `delete` zerstört! Von dem Problem, dass `m == this` sein könnte, will ich gar nicht erst reden.

Eine andere Möglichkeit, die Zuweisung exception-sicher zu gestalten, ist ein »swap-Trick«. Dazu wird eine temporäre Kopie erzeugt, und anschließend werden die Verwaltungsdaten vertauscht. Danach hat `*this` die richtigen Daten, und das temporäre Objekt wird korrekt vom Destruktor zerstört:

```
// exception-sicher
MeinString& MeinString::operator=(MeinString temp) { // Zuweisung
        // temporäre Kopie durch Übergabe per Wert
   // Nutzung der C++-Bibliothek, statt swap() selbst zu schreiben
   std::swap(temp.start, start);
   std::swap(temp.len, len);
   return *this;
}
```

Dieses Vorgehen ist bereits von Seite 218f. bekannt. Falls bei der Bildung von `temp` eine Exception vom Kopierkonstruktor geworfen würde, kämen die Folgezeilen nicht zur Ausführung, und das Objekt auf der linken Seite der Zuweisung bliebe unverändert. Dieses Muster lässt sich auf jede Klasse übertragen. Sie muss nur eine passende `swap()`-Methode besitzen:

```
// exception-sicherer Zuweisungsoperator
Klasse& Klasse::operator=(Klasse kopie) { // temporäre Kopie per Wert
   swap(kopie);              // wirft keine Exception
   return *this;
}
```

20.4 Aussagefähige Fehlermeldung ohne neuen String erzeugen

Exceptions sollen in bestimmten Fällen keine *neuen* Strings erzeugen, um Folge-Exceptions zu vermeiden – das könnte zum Beispiel bei nicht mehr ausreichendem Speicher geschehen. Man kann natürlich Fehlermeldungen statisch ablegen. Das ist jedoch manchmal nicht ausreichend, um eine genaue Information über die Fehlerursache zu erhalten. In solchen Fällen ist es zweckmäßig, einen Puffer vorzuhalten, der mit den Informationen über den Fehlerkontext gefüllt wird. Dies sei an einem praktischen Beispiel gezeigt. Dabei soll eine Exception den allgemeinen Text »Datei kann nicht geöffnet werden:« vorsehen, der um den Fehlerkontext, hier den Namen der betroffenen Datei, ergänzt wird. Das folgende Listing zeigt die Exception-Klasse:

Listing 20.6: Exception-Klasse mit kontextbezogener Fehlermeldung

```
// cppbuch/k25/binaer/IOException.h
#ifndef IOEXCEPTION_H
#define IOEXCEPTION_H
#include<string>
#include<cstring>
#include<exception>

namespace { // anonymer Namespace - nur in dieser Datei gültig
   char msg[100] = "Datei kann nicht geöffnet werden: ";
   size_t msglen = 0;
}

class IOException : public std::exception {
 public:
   IOException(const std::string& filename) {
      if(msglen == 0) { // nur einmal berechnen!
         msglen = strlen(msg);
      }
      int maxRestlaenge = sizeof(msg) - msglen - 1;
      if(maxRestlaenge > 0) { // Puffer um Kontext ergänzen
         strncpy(msg+msglen, filename.c_str(), maxRestlaenge);
      }
   }
   const char* what() const noexcept {
      return msg;
   }
};
#endif
```

Wenn nun ein Programm nach einem IO-Fehler die Anweisung `throw IOException(dateiname);` ausführt, geschieht Folgendes:

- Die Endposition `msglen` des allgemeinen Textes wird gegebenenfalls ermittelt.
- Dann wird der restliche verfügbare Platz ermittelt, der 0 oder positiv sein kann. Die -1 berücksichtigt das abschließende Null-Byte.

- Der Puffer wird um den Namen der betroffenen Datei ergänzt. Die Funktion `strncpy()` (siehe Seite 880) sorgt dafür, dass die Puffergröße nicht überschritten wird.
- Das Anwendungsprogramm kann den Fehlertext mit der redefinierten Methode `what()` holen.

Dem Vorteil, dass keinerlei neuer Speicher angelegt wird und somit durch Speichermangel verursachte Folge-Exceptions unmöglich sind, stehen zwei Schwächen gegenüber, die bei der Anwendung zu berücksichtigen sind:

- Der Rückgabewert von `what()` bleibt nur bis zur nächsten `IOException` gültig, weil die statischen Daten in `msg` bei jeder `IOException` neu überschrieben werden.
- Die Klasse ist nicht thread-sicher, wie auch viele Klassen und Funktionen der Standardbibliothek. Wenn zwei verschiedene Threads gleichzeitig eine `IOException` werfen sollten, ist der Inhalt der Fehlermeldung undefiniert.

In den meisten Fällen spielen diese beiden Punkte keine Rolle.

20.5 Empfehlungen zur Thread-Programmierung

20.5.1 Warten auf die Freigabe von Ressourcen

Verwenden Sie die Konstruktionen

```
while(ressourcenNochNichtBereit) {
   cond.wait(lock);
}
```

mit einer Bedingungsvariablen `cond` und einem Lock-Objekt `lock`. Eine Begründung finden Sie auf Seite 433 und davor auch ein Beispiel. Die Alternative

```
while(ressourcenNochNichtBereit) {
   sleep(zeitdauer);
}
```

sollte nur genommen werden, wenn sich die Variante mit `wait()` als schwierig erweist; solche Fälle gibt es. Keinesfalls sollten Sie

```
while(ressourcenNochNichtBereit) {
}
```

schreiben – es würde sinnlos CPU-Zeit verbraten. Warum wird oben nicht

```
if(ressourcenNochNichtBereit) { // nicht empfehlenswert
   cond.wait(lock);
}
```

statt `while` gewählt, mögen Sie sich fragen. Der Grund liegt darin, dass *nach* der Rückkehr von `wait()`, aber noch *vor* der schließenden Klammer, ein konkurrierender Thread das Flag `ressourcenNochNichtBereit` wieder gesetzt haben könnte. Im Fall der `if`-Anweisung

würde dieses nicht mehr überprüft mit der möglichen Folge, dass zwei Threads gleichzeitig verändernd auf die Ressource zugreifen – ein Fehler!

20.5.2 Deadlock-Vermeidung

Das bekannteste Deadlock-Beispiel in der Informatik ist vermutlich das Problem der »Fünf Philosophen«, die mangels Koordination zu verhungern drohen. Falls Sie das Problem nicht kennen sollten, starten Sie am besten eine Internet-Suche mit dem Begriff »Fünf Philosophen«. Auf Seite 448 wird auf die Gefahr von Verklemmungen (Deadlocks) hingewiesen. Um die dort beschriebene Art von Deadlocks (es gibt leider noch andere) zu vermeiden, gibt es Empfehlungen für die Akquirierung und Freigabe von Ressourcen:

- Wenn mehr als eine Ressource angefordert wird, sollen alle beteiligten Threads sie in derselben Reihenfolge anfordern. Anders gesagt: Wenn ein Thread erst *A* und dann *B* benötigt, gilt diese Reihenfolge für alle Threads.
- Die zuletzt angeforderten Ressourcen sind zuerst wieder freizugeben. Wenn die Anforderung nach dem RAII-Prinzip[2] geschieht, ist automatisch für die richtige Reihenfolge der Freigabe gesorgt.

Der erste Punkt ist nicht immer leicht zu bewerkstelligen. Sehen Sie sich dazu das einfache Beispiel der Überweisung von einem Konto auf das andere aus der Einführung an (Seite 30):

k1.überweisen(54688490, 1000.00)

Die Funktion muss während des Überweisungsvorgangs sicherstellen, dass die Zugriffe von anderen Threads auf beide beteiligte Konten gesperrt werden, etwa auf diese Art:

```
void ueberweisen(long zielkontoNr, double betrag) { // fehlerhaft!
   lock(this);            // eigenes Konto für andere sperren
   Konto* pk = getKonto(zielkontoNr);
   lock(pk);              // Ziel-Konto für andere sperren
   abheben(betrag);       // eigentliche Transaktion
   pk->einzahlen(betrag);
   release(pk);
   release(this);
}
```

Das Problem: Es könnte sein, dass *gleichzeitig* ein Betrag von dem anderen Konto k2 auf das Konto k1 überwiesen werden soll:

k2.überweisen(12573001, 20.95)

In diesem Fall könnten beide Threads gleichzeitig `lock(this)` ausführen – kein Problem, da es sich um verschiedene Objekte handelt. Aber dann bleiben sie aufeinander wartend in der Zeile `lock(pk)` hängen. In solchen Fällen kann eine Reihenfolge für alle vorgegeben werden. Im Beispiel oben wäre es zweckmäßig, das Konto mit der jeweils kleineren Kontonummer zuerst zu sperren.

20.5.3 notify_all oder notify_one?

`notify_one()` befreit *einen* auf die Ressource wartenden Thread, `notify_all()` *alle*. Im letzten Fall kommt nur einer zum Zuge, die anderen werden wieder in die Warteschlange

[2] Siehe Glossar

geschickt, sofern nur ein Thread weiterarbeiten darf. Das muss nicht immer so sein: Zum Beispiel muss ein verändernder Zugriff auf einen kritischen Bereich von nur einem Thread ausgeführt werden. Ein rein *lesender* Zugriff kann daher von vielen Threads gleichzeitig erfolgen, sobald der schreibende Thread die Änderungen vollendet hat. Dann ist ein `notify_all()` angebracht, wobei allerdings Lesen und Schreiben durch verschiedene Bedingungsvariablen gesteuert werden müssen, um die verschiedenen Freigabemöglichkeiten zu unterscheiden.

`notify_one()` kann Anwendung finden, wenn höchstens *ein* Thread die Ressource weiter nutzen soll *und* wenn jeder Thread die Ressource auf dieselbe Art nutzt *und* es nur eine Bedingung gibt, die ggf. zum Warten führt.

In allen anderen Fällen ist `notify_all()` angebracht.

Bei nicht ganz korrekter Programmierung ist `notify_all()` fehlertoleranter. Ein Beispiel:

1. Thread A blockiert einen Datenspeicher begrenzter Kapazität, um die Daten zu sortieren. Der Datenspeicher sei voll, es kann nichts mehr hinzugefügt werden.
2. Thread B möchte Daten entnehmen; damit würde Platz geschaffen. B muss aber warten, weil A noch nicht fertig ist.
3. Thread C möchte Daten in den Datenspeicher schreiben, kann es aber nur, wenn auch Platz vorhanden ist. Auch C wartet.
4. Thread A ist endlich fertig und sagt `notify_one()`. Vom Laufzeitsystem wird genau ein beliebiger Thread aus den Wartenden ausgewählt, in diesem Fall zufällig C.
5. Thread C kann nichts ausrichten und muss warten. Thread B könnte die Lage entschärfen, wartet aber immer noch.

Wenn Thread A in Punkt 4 `notify_all()` gesagt hätte, wäre B zum Zuge gekommen und hätte seinerseits mit `notify_all()` Thread C mitteilen können, dass der Schreibvorgang nunmehr möglich ist.

Tipp

Verwenden Sie im Zweifel lieber `notify_all()` statt `notify_one()`.

`notify_all()` ist bei vielen wartenden Threads natürlich aufwendiger. Andererseits ist der Aufwand, verglichen mit dem des Threads selbst, oft vernachlässigbar. `notify_all()` und `notify_one()` wirken nur im Moment des Aufrufs; er wird nicht gespeichert. Wenn aufgrund eines Programmfehlers ein `notify_all()` oder `notify_one()` ausgeführt wird, der Thread aber erst danach das zugehörige `wait()` erreicht, wird der Thread nie erlöst.

20.5.4 Performance mit Threads verbessern?

Generell kann die Performance einer Applikation mit Threads verbessert werden, wenn es mehr als einen Prozessor(kern) gibt. Dabei ist jedoch zu berücksichtigen, dass Threads Software komplexer und damit fehleranfälliger werden lassen, und dass die Thread-Verwaltung selbst einen bestimmten Aufwand erfordert. Dieser Aufwand soll den Performance-Gewinn nicht überschreiten.

Tipp 1

Parallelisieren Sie ein Programm mit Threads nur dann, wenn Sie wissen, dass es sich wirklich lohnt, nachgewiesen zum Beispiel mit Laufzeitmessungen an einem Prototypen.

Tipp 2 (für Fortgeschrittene)

Der Trend geht zu mehr Prozessorkernen. Wenn Sie zu schreibende Anwendungen davon profitieren lassen wollen und bereit sind, sich in die nicht immer einfache Materie einzuarbeiten, lohnt sich ein Blick auf OpenMP (*http://openmp.org*). OpenMP ist eine Multi-Plattform-Bibliothek zur Parallelisierung von Programmen und zur Kommunikation über shared Memory. Sie wird von vielen Compilern unterstützt.