

20.2.12 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

In C++ kann ein Problem auftreten, wenn eine Ressource beschafft werden soll. 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 827) soll die Erzeugung des Zeigers mit `new` stets innerhalb der Parameterliste geschehen.

```
Ressource pr = new Ressource(id);
shared_ptr<Ressource> spr(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 `spr` wird der Destruktor für `*pr` auch aufgerufen, dann also insgesamt *zweimal*.
- Es könnte sein, dass schon bei der Erzeugung eines Objekts mit `new` eine Exception auftritt. Der damit verbundenen Sprung des Programmablaufs aus dem aktuellen Kontext zum Beispiel zu einer `catch`-Klausel führt dazu, dass ein `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 verlas-

senen 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 827). Dies kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 201 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`-Objekt 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 Exception-sichere Funktion

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

20.3.4 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.

Nehmen wir an, dass ein Vektor nur eine maximale Größe von 64000 haben soll, und ziehen dazu die Vektor-Klasse von Seite 320 heran, deren Konstruktor verändert wird:

```
template<typename T>
inline Vektor<T>::Vektor(size_t x)
: xDim(x), start(0){
    if(x > 64000) {
        throw std::length_error("Vektor: Dimension zu groß!");
    }
    else {
        start = new T[x];
    }
}
```

Die mögliche Verwendung:

```
int main() {
    try {
        Vektor<int> v(64001); // zu groß! Exception
        // ... geplante Verwendung des Vektors
    } // Destruktor ~Vektor() wird aufgerufen
    catch(const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
}
```

Im main-Programm wird so festgestellt, dass etwas schief gegangen ist. Jetzt nehme ich aber an, dass zwar die Größe unter 64000 liegt, aber dann etwas bei der Speicherbeschaffung schief gegangen ist:

```
int main() {
    try {
        Vektor<int> v(10000); // Annahme: interner Fehler bei new T[x]
        // ... geplante Verwendung des Vektors
    } // Destruktor ~Vektor() wird aufgerufen
    catch(const exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
}
```

Der Destruktor `~Vektor()` führt `delete [] start;` aus. Falls aber `start` *nicht* mit 0 initialisiert worden wäre, führte das zum Crash! Die Konsequenz ist der folgende Tipp:



Tipp

Alle Zeiger, denen *im Body* des Konstruktors ein mit `new` erzeugtes Objekt zugewiesen werden soll, müssen in der Initialisierungsliste des Konstruktors mit `NULL` initialisiert werden!

Die Alternative ist (wie bisher gehandhabt) die Zuweisung von Speicher in der Initialisierungsliste, also

```
template<typename T>
inline Vektor<T>::Vektor(size_t x)
: xDim(x), start(new T[x]){
    if(x > 64000) {
        throw std::length_error("Vektor: Dimension zu groß!");
    }
}
```

20.3.5 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 Zuweisungsoperator-Aufgabe von Seite 326 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;
}
```

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
// korrigiert entspr. www.cppbuch.de/errata.html
Klasse& Klasse::operator=(Klasse kopie) { // temporäre Kopie per Wert
    swap(kopie); // wirft keine Exception
    return *this;
}
```