

```
...
}
```

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/exclmKonstruktor1.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;
}

```