21.2.16 One Definition Rule

Jede Variable, Funktion, Struktur, Konstante und so weiter in einem Programm hat *genau eine* Definition.

21.2.17 Defensiv Objekte löschen

Hier geht es um die mögliche Fehlerursache, 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 == nullptr) {                      // Fehlerhafte Annahme
...
}
```

Der Fehler liegt in dem undefinierten Wert von `pa` nach der Löschoperation. Falls ein Zeiger nach dem Löschen noch verwendet werden kann, setzen Sie ihn direkt nach dem `delete` mit `pa = nullptr;` auf Null. Dann kann er geprüft werden und es gibt eine definierte Fehlermeldung. Besser noch ist jedoch die Vermeidung solcher Konstruktionen zugunsten der Kapselung von `new` und `delete` oder der Verwendung von `unique_ptr` bzw. `shared_ptr`, siehe folgenden Abschnitt.

21.2.18 Speicherbeschaffung und -freigabe kapseln

Die Operatoren `new` und `delete` sind stets paarweise zu verwenden. Um Speicherfehler zu vermeiden, empfiehlt sich das »Verpacken« dieser Operationen in Konstruktor und Destruktor wie bei der Vektorklasse des Kapitels 8 oder die Verwendung der »Smart Pointer« (`shared_ptr`), siehe unten. Ein weiterer Vorteil ist die korrekte Speicherfreigabe bei Exceptions (siehe unten).

21.2.19 Programmierrichtlinien einhalten

Das Einhalten von Programmierrichtlinien unterstützt das Schreiben gut lesbarer Programme. Es gibt einige dieser Richtlinien, die sich in großen Teilen ähneln. Oft hat eine Software-entwickelnde Firma eine eigene Richtlinie.

21.3 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.

21.3.1 Sichere Verwendung von unique_ptr und shared_ptr

Bei der Konstruktion eines `unique_ptr` bzw. `shared_ptr` (Beschreibung in Kapitel 32) 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). Entsprechendes gilt für `unique_ptr`. Noch besser, weil einfacher, ist die gänzliche Vermeidung von `new`, wie der folgende Abschnitt zeigt.

21.3.2 So vermeiden Sie new und delete!

Wie gezeigt, muss man sich um `delete` nicht mehr kümmern, wenn `unique_ptr` oder `shared_ptr` eingesetzt werden. Die Hilfsfunktionen `make_unique` (siehe Abschnitt 32.1.1) und `make_shared` (siehe Abschnitt 32.2.1) vereinfachen die Schreibweise weiter, sodass auch `new` entfällt. Dabei werden nur noch die Argumente für den Konstruktor übergeben. Im folgenden Beispiel benötigt der Konstruktor nur ein `int`-Argument:

```
// vector mit shared_ptr
std::vector<std::shared_ptr<Ressource>> vec1(10);
vec1[0] = std::shared_ptr<Ressource>(new Ressource(1));          // mit new
// einfacher ist:
vec1[0] = std::make_shared<Ressource>(1);                        // ohne new

// vector mit unique_ptr
std::vector<std::unique_ptr<Ressource>> vec2(10);
vec2[0] = std::unique_ptr<Ressource>(new Ressource(2));          // mit new
// einfacher ist:
vec2[0] = std::make_unique<Ressource>(2);                        // ohne new
```

C++14

Die Zuweisung im zweiten Beispiel ist nur möglich, weil auf der rechten Seite ein R-Wert (temporäres Objekt) steht. Wäre es nicht temporär, gäbe es eine Fehlermeldung des Compilers. Beispiel:

```
auto uniqueptr999 = std::make_unique<Ressource>(999);
vec2[0] = uniqueptr999;                                    // Fehler!
```

Damit wird verhindert, dass es zwei `unique_ptr`-Objekte geben kann, die auf dasselbe Heap-Objekt verweisen. Eine Kopie ist nicht erlaubt.

21.3.3 So vermeiden Sie new[] und delete[]!

Nach `new[]` nur `delete` statt `delete[]` zu schreiben, wäre ein Fehler. Er ist leicht zu vermeiden, wenn auf `new[]` zugunsten von `vector` verzichtet wird. In den meisten Fällen wird das ohne Probleme möglich sein. Ein Beispiel dafür ist die String-Klasse von Seite 247. Manche empfehlen die Verwendung von `unique_ptr<T>` (siehe unten). Innerhalb einer Klasse, deren Objekte kopierbar sein sollen und für die Speicherplatz beschafft werden soll, würde man nur den Destruktor sparen, nicht aber den Kopierkonstruktor und Zuweisungsoperator. Die Verwendung von `vector` spart auch diese ein.

21.3.4 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. Dies kann zu einem Speicherleck führen, wenn der Zeiger mit `new []` erzeugt wurde, wie auf Seite 214 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 ist die Übergabe eines `deleter`-Objekts an den `shared_ptr`-Zeiger. Wenn es so ein Funktionsobjekt gibt, wird dessen `operator()()` aufgerufen.

```
// Auszug aus cppbuch/k32/arrayshared.cpp
template<typename T>
struct ArrayDeleter {
   void operator()(T* ptr) {
      delete [] ptr;
   }
};
```

```
void funktion() {              // mögliche Anwendung
   shared_ptr<int> p(new int[10], ArrayDeleter<int>());    // richtig
```

```
   // ... etliche Zeilen weggelassen
}                                          // kein Memory-Leak, Array wird korrekt gelöscht
```

Tipp

Sie können einen vergessenen Deleter leicht vermeiden, wenn Sie auf `shared_ptr` für Arrays verzichten und stattdessen einen `shared_ptr` mit einem `vector` verwenden, etwa so `auto ptr = std::make_shared<std::vector<int>>();`, siehe auch Abschnitt 21.3.2 oben.

21.3.5 unique_ptr für Arrays korrekt verwenden

Das Problem des vergessenen Deleters tritt bei `unique_ptr` nicht auf, weil der Typ des für die Löschung zuständigen Objekts zur Schnittstelle gehört.

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

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 21.1: `unique_ptr` und Array

```
// cppbuch/k32/arrayunique.cpp
#include<memory>

void f() {
   std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
   // entspricht std::unique_ptr<int[]> arr(new int[10]);
   // Benutzung des Arrays weggelassen
}    // kein Memory-Leak, Array wird korrekt gelöscht

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

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

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

Tipp

Im Verzeichnis *cppbuch/k12/move/unique_ptr* finden Sie ein Beispiel für einen String-Typ, der einen `unique_ptr` auf ein `char`-Array verwendet. Überlegen Sie sich bei einem ähnlichen Problem aber, ob nicht doch ein `vector` die einfachere Lösung ist.

21.3.6 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 sollen ausschließlich Stack-Objekte (automatische Objekte) verwendet werden, wenn Exceptions auftreten. Dies ist immer möglich, wenn Beschaffung und Freigabe eines dynamischen Objekts innerhalb eines Stack-Objekts versteckt werden. Das Hilfsmittel dazu kennen wir bereits, nämlich die »intelligenten« Zeiger aus Abschnitt 8.5:

```
void func() {
     // shared_ptr der Standardbibliothek, siehe Abschnitt 8.5.1. 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.

21.3.7 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. »Vollständig erzeugt« heißt, 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! Die Klasse `Ganzes` folgt im Anschluss.

Listing 21.2: Klassen `Teil1` und `Teil2`

```
// cppbuch/k21/teil.h
#ifndef TEIL_H
#define TEIL_H
#include<iostream>

class Teil1 {
public:
   Teil1(int x)
      : attr(x) {
   }
   ~Teil1() {
      std::cout << "Teil1::Destruktor gerufen!\n";
   }
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!\n";
   }
};
#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 21.3: Klasse `Ganzes`

```
// cppbuch/k21/ganzes.h
#ifndef GANZES_H
#define GANZES_H
#include<iostream>
#include"teil.h"

class Ganzes {
public:
   Ganzes() : teil1(99) {
      ptr = new Teil2;
   }
   ~Ganzes() {
```

```
      std::cout << "Ganzes::Destruktor gerufen!\n";
      delete ptr;
   }
private:
   Teil1 teil1;
   Teil2* ptr;
   Ganzes(const Ganzes&) = delete;                  // für Beispiel nicht erforderlich
   Ganzes& operator=(const Ganzes&) = delete;       // für Beispiel nicht erforderlich
};
#endif
```

Listing 21.4: Exception im Konstruktor

```
// cppbuch/k21/excImKonstruktor1.cpp
#include"ganzes.h"

int main() {
   try {
      Ganzes ganzes;
   }
   catch(const std::exception& e) {
      std::cerr << "Exception gefangen: " << e.what() << '\n';
   }
}
```

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 21.5: Fehlerhafter Konstruktor

```
// Auszug aus cppbuch/k21/ganzesMitFehler.h
class GanzesMitFehler {
public:
   GanzesMitFehler() : ptr1(new Teil1(99)), ptr2(new Teil2) {
   }
   ~GanzesMitFehler() {
      std::cout << "GanzesMitFehler::Destruktor gerufen!\n";
      delete ptr1;
      delete ptr2;
   }
private:
   Teil1* ptr1;
   Teil2* ptr2;
   // Kopierkonstruktor und Zuweisungsoperator weggelassen
};
```

Mit `shared_ptr` wie in Abschnitt 21.3.1 geht man sicher. Wenn der `Teil2`-Konstruktor eine Exception wirft, wird der Destruktor von `Teil1` aufgerufen und gibt den Speicherplatz frei:

Listing 21.6: Konstruktor mit `shared_ptr`

```
// Auszug aus cppbuch/k21/ganzesKorrigiert.h
class GanzesKorrigiert {
public:
   GanzesKorrigiert() : ptr1(new Teil1(99)), ptr2(new Teil2) {
   }
   ~GanzesKorrigiert() {
      std::cout << "GanzesKorrigiert::Destruktor gerufen!\n";
       // delete nicht notwendig wegen shared_ptr
   }
private:
   std::shared_ptr<Teil1> ptr1;
   std::shared_ptr<Teil2> ptr2;
};
```

21.3.8 Exception-sichere Zuweisung

Wenn bei einer Kopie Speicher beschafft werden muss, ist es zuerst zu erledigen! 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 andere Lösung für den Zuweisungsoperator von Seite 350 an:

```
template<typename T>                                    // Exception-sicher
Vektor<T>& Vektor<T>::operator=(const Vektor<T>& v) {   // Zuweisung
   T* temp = new T[v.anzahl];                          // zuerst neuen Platz beschaffen
   for(std::size_t i = 0; i < v.anzahl; ++i) {                  // kopieren
      temp[i] = v.start[i];
   }
   delete [] start;                                  // alten Platz freigeben
   anzahl = v.anzahl;                                // Verwaltungsinformation aktualisieren
   start = temp;
   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 `temp` nicht:

```
template<typename T>                                    // NICHT Exception-sicher!
Vektor<T>& Vektor<T>::operator=(const Vektor<T>& v) {   // Zuweisung
   delete [] start;                                  // weg damit, es wird schon gutgehen!
   start = new T[v.anzahl];                                  // neuen Platz beschaffen
   for(std::size_t i = 0; i < v.anzahl; ++i) {               // kopieren
      start[i] = v.start[i];
   }
```

```
   anzahl = v.anzahl;                              // 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 `&v == this` sein könnte, will ich gar nicht erst reden.

Der »swap-Trick« liefert eine noch bessere Möglichkeit, die Zuweisung exception-sicher zu gestalten. Sie sahen ihn bereits auf Seite 350. 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;
}
```

21.4 Empfehlungen zur Thread-Programmierung

21.4.1 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 491 und davor auch ein Beispiel. Die Alternative

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

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