Strukturierte Datentypen und Klassen

☑ **Lösung zu Aufgabe 3.1** Die erhaltene Ausgabe ist `d-c d-c c-c d c-c d a d`, wobei wir der Übersichtlichkeit halber die einzelnen Ausgaben abgekürzt haben, beispielsweise wurde `copy-construct` zu `c-c` vereinfacht.

Speziell zu erwähnen ist das Objekt `o3`. Zum einen handelt es sich bei dessen Konstruktion tatsächlich um einen Aufruf des Kopierkonstruktors (`c-c`) und nicht, wie man wegen dem `=` denken könnte, um einen Aufruf des Zuweisungsoperators (`a`). Zum anderen wird `o3` unmittelbar nach seiner Konstruktion wieder zerstört (`d`), weil es als Stackvariable in einem eigenen Scope definiert ist.

Ebenso erwähnenswert ist die Zuweisung `o4 = &o1`. Da es sich hierbei nur um eine Zuweisung von Zeigern handelt, wird keine der vier definierten Klassenmethoden aufgerufen und entsprechend produziert diese Zeile gar keine Ausgabe.

Zu guter Letzt beobachten wir, dass insgesamt vier Konstruktoren (`d-c` und `c-c`) aufgerufen werden, aber nur drei Destruktoren (`d`). Dies liegt daran, dass zwar zwei Objekte mit dem Schlüsselwort `new` auf dem Heap erstellt werden, aber nur eines davon mit dem Schlüsselwort `delete` wieder gelöscht wird.

☑ **Lösung zu Aufgabe 3.2** Die Klasse `std::unique_ptr<int>` funktioniert nach dem allgemeinen RAII-Prinzip: In den Konstruktoren wird eine Ressource in Besitz genommen und im Destruktor wird die Ressource wieder freigegeben. Im konkreten Fall von `std::unique_ptr<int>` bedeutet dies, dass der Konstruktor die Adresse eines Heap-allozierten `int` in einer Member-Variable speichert, wodurch effektiv Besitz von dem referenzierten `int` ergriffen wird. Der Destruktor wiederum entfernt durch Verwendung des Schlüsselworts `delete` ebendiesen `int` vom Heap. Dadurch wird gewährleistet, dass der vom `int` verwendete Speicher automatisch freigegeben wird, sobald sein Besitzer (d.h. das Objekt vom Typ `std::unique_ptr<int>`) zerstört wird.

Der Kopierkonstruktor und der Zuweisungsoperator sind für die Klasse `std::unique_ptr<int>` nicht sinnvoll implementierbar und deshalb existieren sie auch nicht in der Standardbibliothek. Die grundlegende Idee hinter der Klasse ist, dass der referenzierte `int` einen einzigen (deshalb *unique*) Besitzer haben soll. Wenn durch Erstellung einer Kopie aber plötzlich zwei Objekte vom Typ `std::unique_ptr<int>` auf denselben `int` zeigen würden, dann hätte dieser `int` effektiv zwei Besitzer.

☑ **Lösung zu Aufgabe 3.3** Mit einer Member-Variable `m_name` kann sich jedes `ScopeLogger`-Objekt den Namen des eigenen Scopes merken. Dann muss man nur noch dafür sorgen, dass beim Betreten eines Scopes (d.h. wenn der Konstruktor aufgerufen wird) und beim Verlassen des Scopes (d.h. wenn der Destruktor aufgerufen wird) die gewünschten Meldungen ausgegeben werden.

Für die Ausgabe der Punkte merken wir uns zusätzlich in einer statischen Klassen-Variable `depth` die aktuelle Verschachtelungstiefe (d.h. wieviele verschiedene Scopes momentan aktiv sind). Entsprechend muss diese Variable im Konstruktor inkrementiert und im Destruktor dekrementiert werden.

```
class ScopeLogger {
private:
  const char* m_name;
  static int depth;
public:
  ScopeLogger(const char* name) : m_name(name) {
    ++depth;
    std::cout << std::string(depth, '.')
              << " Enter scope " << m_name << std::endl;
  }
  ~ScopeLogger() {
    std::cout << std::string(depth, '.')
              << " Leave scope " << m_name << std::endl;
    --depth;
  }
};

int ScopeLogger::depth = 0;
```