C++: Resource Acquisition Is Initialization

16. Dezember 2012

Hintergrund

- Viele haben C++ als C mit Klassen kennen gelernt
- Exceptions erfordern anderes Aufräumen als in C
- Durch neue Features von C++11 RAII und implizite Speicherverwaltung angenehmer
- Ziel: niemals manuell new und delete aufrufen
- Ziel: keine raw Pointer erzeugen oder an Methoden weiterreichen
- Ziel: dynamische Speicherverwaltung durch RAII-Objekte realisieren

Semantik: Value, Move

- Value-Semantik
  - Jede Klasse und jedes Struct kann sich wie ein primitiver Datentyp verhalten
  - Andere Sprachen kennen nur primitive Datentypen und Referenzen
  - Statt Referenzen liegen in C++ Objekte auf dem Stack einer Methode
- Move-Semantik (C++11)
  - Nur Value-Semantik würde viel Kopieren bedeuten
  - std :: vector<**int**> x = bla()
  - Neben & für lvalue-Referenzen gibt es jetzt && für rvalue-Referenzen
  - std :: vector<T>::**operator**=(std::vector<T>&& rhs)
  - rhs steht kurz vor der Zerstörung und operator=() ist der letzte Nutzer

Das Problem

```
int * ip = new int[100]; // setup()
bla (); // do_something()
delete [] ip; // cleanup()
```

- Durch Templates und Exceptions ist C++ mehr als C mit Klassen
- Jede Funktion kann eine Exception werfen, kein finally
- Exceptions können Programmfluss jederzeit unterbrechen
- C++ zerstört Stackvariablen beim Verlassen einer Funktion
- Neueste Stackvariable zuerst gelöscht
- Destruktor vom Pointer ungleich dem des Arrays

Die Lösung (für das Array)

```
struct Array {
  int * ip;
  Array(size_t size) : ip(new int[size]) {}
  ~Array() {
    delete [] ip;
  }
};

Array a(100);
bla();
```

- Egal was in bla() passiert, der Speicher wird leer geräumt
- Viele STL Klassen wie std::vector, std::string oder std::fstream sind so aufgebaut

unique_ptr (C++11)

```
std :: uniqueptr<int> i(new int);
std :: uniqueptr<int> x = i; // compilerfehler
std :: uniqueptr<int> y = std::move(i); // geht
```

- Funktion hat Pointer als Rückgabewert, unklar wer delete aufruft
- C++11: auto_ptr durch shared_ptr, weak_ptr, unique_ptr ersetzt
- Stellt sicher, dass nur ein Zeiger auf ein Objekt existiert
- Kann mit move-Semantik verschoben werden

unique_ptr mit C-Libs (C++11)

```
struct mat;
mat * mat_alloc( size_t  size );
void mat_free(mat * m);
unique_ptr<mat, void (*)(mat *)> m(mat_alloc(r), mat_free);
```

- C Objekte haben create und free Funktionen
- unique_ptr muss statt delete die Funktion der Bibliothek nutzen

Semaphoren

```
struct Lock {
  Semaphore& sem;
  Lock(Semaphore& sem) : sem(sem) {
    sem.down();
  }
  ~Lock() {
    sem.up();
  }
};

Semaphore camera, machine_gun;
Lock l_camera(camera), l_machine_gun(machine_gun);
focus();
fire();
```