Advanced Software Development

Smart Pointer

Motivation

- Speicherverwaltung in C++ ist fehleranfällig
- Memory Leaks durch vergessenes delete
- Dangling Pointers durch verfrühtes delete
- Besonderes Problem: Exception Safety

Bsp.: Exceptions

```
void doSomething(int x) {
  if (x < 0) {
    throw NegativeValueException();
  }
  else {
    …
  }
}

void doSomethingDifferent() {
  Object *obj = new MyObject;
  …
  doSomething(obj->getX());
  delete obj;
}
```

Exception Safety

- Freigabe aller belegter Ressourcen auch im Falle von Exceptions
- Lösungsmöglichkeiten
  - finally-Block (nicht in C++)
  - Ausnahme abfangen und weitergeben
  - Kapselung in Klasse mit Konstruktor/Destruktor (RAII)

RAII Technik

- RAII = Resource Acquisition is Initialization („Ressourcenbelegung ist Initialisierung)
- Belegung im Konstruktor
- Freigabe im Destruktor
- Impliziter Aufruf des Destruktors (auch im Fall einer Exception)

Kapselung in einer Template-Klasse

```
template <typename T>
class SmartPtr {
public:
  SmartPtr(T *p = nullptr) : ptr(p) {}
  ~SmartPtr() { delete ptr; }

private:
  T *ptr;
};


void doSomethingDifferent(…) {
  Object *obj = new MyObject;
  SmartPtr<Object> pObj(obj);
  …
  doSomething(obj->getX());
}
```

Smart Pointer Klasse (1)

```
template <typename T>
class SmartPtr {
public:
  SmartPtr(T *p = nullptr) : ptr(p) {}
  ~SmartPtr() { delete ptr; }

  // forbid copy constructor and assignment
  SmartPtr(SmartPtr const &) = delete;
  SmartPtr &operator=(SmartPtr const &) = delete;

  T *get() const {
    return ptr;
  }
```

Smart Pointer Klasse (2)

```
  void reset(T *p = nullptr) {
    assert(p == nullptr || p != ptr);  // catch self-reset errors
    delete ptr;
    ptr = p;
  }

  T &operator*() const {
    assert(ptr != nullptr);
    return *ptr;
  }

  T *operator->() const {
    assert(ptr != nullptr);
    return ptr;
  }

private:
  T *ptr;
};
```

std::auto_ptr

- Smart Pointer in der C++ Standard Library (<memory> Header)
- Beispiel:

```
void doSomethingDifferent(…) {
  auto_ptr<Object> obj(new MyObject);
  …
  doSomething(obj->getX());
}
```

std::auto_ptr – Besonderheiten

- Kopieren und Zuweisung haben Move-Semantik (=> „Transfer of Ownership“)

```
auto_ptr<Object> obj1(new MyObject);
auto_ptr<Object> obj2(obj1);  // obj1.get() == 0
obj1 = obj2;                  // obj2.get() == 0
```

- Zu kopierendes Objekt wird verändert!
- „Kopieren“ verhindern:

```
auto_ptr<Object> const obj1(new MyObject);
auto_ptr<Object> obj2(obj1);  // nicht erlaubt!
```

- „Release of Ownership“:

```
auto_ptr<Object> obj(new MyObject);
Object *pObj = obj.release(); // keine automatische Freigabe mehr
```

std::unique_ptr

- Neu in C++ 11 (Header <memory>)
- Macht std::auto_ptr obsolet
- Korrekte Move-Semantik mittels Move Constructor bzw. Move Assignment; kein Copy Constructor bzw. Copy Assignment
- Beispiel

```
unique_ptr<Object> obj1(new Object);
// unique_ptr<Object> obj2(obj1);     // nicht erlaubt!
unique_ptr<Object> obj2(move(obj1));  // obj1.get() == nullptr
obj1 = move(obj2);                    // obj2.get() == nullptr
auto obj3 = make_unique<Object>();    // unique_ptr<Object>(new Object)
```

std::shared_ptr

- Neu in C++ 11 (Header <memory>)
- Ermöglicht gemeinsame Verwendung von Objekten (mittels Reference Counting)
- Kopieren und Zuweisung möglich (verändert den Reference Count)
- Automatische Freigabe des Objekts bei der Freigabe des letzten shared_ptr (wenn Reference Count = 0)

std::shared_ptr – Beispiel

```
class StringHolder {
public:
  explicit StringHolder(std::shared_ptr<std::string> s) : str(s) {}
  …
private:
  std::shared_ptr<std::string> str;
};

int main() {
  unique_ptr<StringHolder> sh1;
  …
  if (…) {
    shared_ptr<string> s(new string("abc"));  // RefCount=1
    StringHolder sh2(s);                      // RefCount=2
    …
    sh1.reset(new StringHolder(sh2));         // RefCount=3
  }                                           // RefCount=1
  …
}                                             // RefCount=0 => Freigabe
```

std::make_shared

- Erzeugt Objekt und Referenzzähler gemeinsam in einem Speicherblock

```
auto s = make_shared<string>("abc");
// statt shared_ptr<string> s(new string("abc"));
```

shared_ptr – Members (1)

- Konstruktoren
  - Default-Konstruktor (Null-Pointer)
  - Konstruktion aus Pointer
  - Copy Constructor
  - Move Constructor
- Destruktor
- Zuweisungsoperator (Copy/Move)

shared_ptr – Members (2)

- Methoden und Operatoren
  - reset: Ersetzen des Pointers (Default: nullptr)
  - get: Abfragen des Pointers
  - Dereferenzierungs-Operatoren (* und ->)
  - use_count: Abfrage des Reference Count
  - unique: Prüfung, ob einzige Referenz
  - Implizite Konvertierung nach bool
  - swap: Vertauschen mit anderem shared_ptr

std::weak_ptr

- Beobachter (Observer) eines shared_ptr
- keine Auswirkung auf Reference Count
- Erlaubt Prüfung, ob Objekt noch existiert
- Kein Direktzugriff auf Pointer
  - keine get Methode
  - keine Dereferenzierungsoperatoren
  - Zugriff erfordert (temporären) shared_ptr

std::weak_ptr – Beispiel

```
auto obj = make_shared<Object>();
weak_ptr<Object> objObserver(obj);
…
{
  …
  auto lockedObj = objObserver.lock();
  // alternativ: shared_ptr<Object> lockedObj(objObserver);
  if (lockedObj) {
    …
  }
}
obj.reset();
if (objObserver.expired()) {
  cout << "Object not available any more";
}
```

boost::intrusive_ptr

- Bestandteil der Boost Library
- Alternative zu std::shared_ptr
- Ermöglicht das Speichern des Reference Count im Objekt selbst
- Reference Counting Mechanismus muss selbst implementiert werden

boost::intrusive_ptr – Beispiel (1)

```
class RefCounted {
public:

  void incRef() {
    ++mRefCount;
  }

  int decRef() {
    return --mRefCount;
  }

private:
  int mRefCount = 0;
};
```

boost::intrusive_ptr – Beispiel (2)

```
void intrusive_ptr_add_ref(RefCounted *rc) {
  rc->incRef();
}

void intrusive_ptr_release(RefCounted *rc) {
  if (rc->decRef() == 0) {
    delete rc;
  }
}

class HasRefCounted {
public:
  HasRefCounted() : mRefCounted(new RefCounted) {}
  …
private:
  boost::intrusive_ptr<RefCounted> mRefCounted;
};
```

Smart Pointer für Arrays (1)

- std::unique_ptr funktioniert auch für Arrays

```
size_t n = …;
unique_ptr<int[]> arr(new int[n]);
arr[0] = 3;
int *begin = arr.get();
```

Smart Pointer für Arrays (2)

- std::shared_ptr erfordert die Angabe eines Custom Deleters

```
template <typename T>
struct array_deleter {
  void operator ()(T *p) {
    delete [] p;
  }
};

shared_ptr<int> sarr(new int[n], array_deleter<int>());
sarr.get()[0] = 3;  // nicht möglich: sarr[0]
int *begin = sarr.get();
```

Smart Pointer in Containern

- std::unique_ptr, std::shared_ptr und boost::intrusive_ptr können als Elementtypen verwendet werden
- std::auto_ptr kann nicht verwendet werden
- Alternative: Boost Pointer Container

Smart Pointer in Containern: Bsp.

```
using Animals = vector<unique_ptr<Animal>>;

int main() {
  Animals animals;
  animals.emplace_back(new Cat("Tom"));
  animals.emplace_back(new Mouse("Jerry"));
  //…
  cout << animals.size() << " animals:\n";
  cout << animals[0]->getName() << endl;
  for (auto &animal : animals) {
    cout << animal->getName() << endl;
  }
  return 0;
} // automatische Freigabe aller Objekte in animals
```