33

Speichermanagement

Dieses Kapitel behandelt die folgenden Themen:

- Smart Pointer der Standardbibliothek
- new mit vorgegebenem Speicherort
- Hilfsfunktionen

33.1 Smart Pointer unique_ptr, shared_ptr, weak_ptr

Die Wirkungsweise eines Smart Pointers wird in Abschnitt 9.5 beschrieben. Hier geht es um die verschiedenen Realisierungen der C++-Standardbibliothek.

unique_ptr

Die Klasse `unique_ptr` verhält sich wie die Klasse `SmartPointer` des Abschnitts 9.5, hat aber zusätzliche Funktionen. So kann zum Beispiel ein Objekt übergeben werden, das die Zerstörung anstelle des normalen Destruktors übernimmt. Da der Letztere im Allgemeinen genügt, wird hier auf eine Darstellung verzichtet. Ein einfaches Beispiel:

Listing 33.1: unique_ptr

```
// cppbuch/k33/uniqueptr/main.cpp
#include<iostream>
#include<memory>
using namespace std;

class Ressource {
public:
   Ressource(int i)
      : id(i){
      cout << "Konstruktor Ressource()" << endl;
   }
   void hi() const {
      cout << "hier ist Ressource::hi (),  Id=" << id << endl;
   }
   ~Ressource() {
      cout << "Ressource::Destruktor, Id=" << id << endl;
   }
private:
   int id;
};

int main() {
   cout << "Zeiger auf dynamisches Objekt:" << endl;
   unique_ptr<Ressource> p1(new Ressource(1));
   cout << "Operator -> ";
   p1->hi();
   cout << "Operator * ";
   (*p1).hi();
   // Null-Zeiger
   unique_ptr<Ressource> nullp((Ressource*)0);
   // nullp->hi(); // Speicherzugriffsfehler!
}
```

Ausgewählte Methoden:

- `unique_ptr()` erzeugt ein Objekt, das nichts enthält, gleichbedeutend mit der Anweisung `unique_ptr<Ressource> nullp((Ressource*)0);` im obigen Programm.
- `operator->()` gibt den Zeiger auf das enthaltene Objekt zurück.
- `operator*()` gibt eine Referenz auf das enthaltene Objekt zurück, d.h. `*operator->()`.
- `get() const` gibt genau wie `operator->()` den Zeiger auf das enthaltene Objekt zurück.
- `operator bool() const` gibt `get() != NULL` zurück.
- `reset(ptr)` setzt den internen Zeiger auf `ptr`. Der Destruktor für das möglicherweise vorher enthaltene Objekt wird aufgerufen.
- `release()` setzt den internen Zeiger auf `NULL`. Achtung: Der Destruktor für das möglicherweise vorher enthaltene Objekt wird *nicht* aufgerufen! Daher wird es im Allgemeinen besser sein, `reset(NULL)` zu nehmen..

Zur Verwendung von `unique_ptr` für Arrays siehe Abschnitt 20.3.3, Seite 568.

shared_ptr

Die Klasse `shared_ptr` implementiert eine Benutzungszählung. Damit können mehrere Objekte dieser Klasse auf ein Objekt (im Folgenden zur Unterscheidung Ressource genannt) verweisen. Ein `shared_ptr`-Objekt speichert die Adresse einer mit `new` erzeugten Ressource:

```
class X {};
shared_ptr<X> p1(new X);
```

Wenn ein weiteres `shared_ptr`-Objekt mit derselben Adresse initialisiert wird, erhöht sich der interne Benutzungszähler:

```
shared_ptr<X> p2(p1);
cout << p1.use_count() << endl; // 2
```

Der Destruktor der Klasse `shared_ptr` zählt den Benutzungszähler um eins herunter. Der Destruktor des letzten auf die Ressource verweisenden `shared_ptr`-Objekts ist für ihre Zerstörung verantwortlich. Man sagt auch, dass das `shared_ptr`-Objekt die Ressource *besitzt*. Das folgende Beispiel zeigt diese und weitere Eigenschaften von `shared_ptr`:

Listing 33.2: Beispiel mit SmartPointer-Objekten

```
// cppbuch/k33/sharedptr/main.cpp
#include<memory>
#include<iostream>
using namespace std;

class Ressource {
public:
   Ressource(int i)
      : id(i){
   }
   void hi() const {
      cout << "hier ist Ressource::hi(),  Id=" << id << endl;
   }

   ~Ressource() {
      cout << "Ressource::Destruktor, Id=" << id << endl;
   }
private:
   int id;
};

int main() {
   cout << "Konstruktoraufruf" << endl;
   shared_ptr<Ressource> p1(new Ressource(1));

   cout << "Operator -> "; p1->hi();
   cout << "Operator * "; (*p1).hi();
   cout << "Benutzungszähler: " << p1.use_count() << endl; // 1

   { // Blockanfang
      // zweiter shared_ptr für dasselbe Objekt
```

```
    shared_ptr<Ressource> p2(p1);
    cout << "Benutzungszähler p1: " << p1.use_count() << endl; // 2
    cout << "Benutzungszähler p2: " << p2.use_count() << endl; // 2
    p2->hi();
  } // p2 wird zerstört
  cout << "Benutzungszähler p1: " << p1.use_count() << endl; // 1
  cout << " Objekt existiert noch: ";
  p1->hi();
  // Zuweisung
  shared_ptr<Ressource> p3(new Ressource(3));
  p3 = p1;  // Ressource 3 wird freigegeben (delete), danach
            // verweisen beide auf das Objekt *p1
  p1->hi();
  p3->hi();
  // Null-Zeiger
  shared_ptr<Ressource> nullp((Ressource*)0);
  // nullp->hi(); // Speicherzugriffsfehler!
} // p3 und p1 werden zerstört
```

Die fehlerhafte Nutzung mit einem Null-Zeiger, wie am Ende gezeigt, führt zum Programmabbruch. Bei richtiger Handhabung entsprechend dem unten beschriebenen Tipp tritt dieser Fall nicht auf. An der abschließenden }-Klammer werden die Destruktoren der noch verbliebenen Objekte `p3` und `p1` ausgeführt. Nur der zuletzt ausgeführte Destruktor löscht das referenzierte `Ressource`-Objekt. Container der Standardbibliothek (siehe Kapitel 28) können statt Zeigern `shared_ptr`-Objekte enthalten:

```
// STL-Container mit shared_ptr
vector<shared_ptr<Ressource> > vec(10);
vec.push_back(p3);  // p3 von oben
vec[1] = shared_ptr<Ressource>(new Ressource(4));
vec[1]->hi();
```

Tipp 1

Wenn Sie dynamische Objekte erzeugen, verwenden Sie `shared_ptr`. Über die Zerstörung mit `delete` an einer geeigneten Stelle müssen Sie sich keine Gedanken mehr machen. Die Erzeugung des Zeigers mit `new` muss innerhalb der Parameterliste geschehen (Begründung siehe Abschnitt 20.3.1 auf Seite 567).

Tipp 2

Wenn Sie `shared_ptr` für Arrays einsetzen, müssen Sie eine Hilfsklasse zur Vermeidung von Memory-Leaks schreiben. Die Einzelheiten finden Sie in Abschnitt 20.3.2, Seite 567.

weak_ptr

`weak_ptr`-Objekte sind für Objekte gedacht, die bereits von `shared_ptr`-Objekten verwaltet werden. Der Konstruktor:

```
template<class T>
weak_ptr(const shared_ptr<T>& ptr);
```

Der Unterschied zu `shared_ptr` ist, dass ein `weak_ptr` kopiert und zugewiesen werden kann. Der `weak_ptr`-Destruktor hat keine Wirkung auf das enthaltene Objekt. Im Gegensatz zu einem `shared_ptr` besitzt ein `weak_ptr` keine Ressource, er verweist nur auf sie. `weak_ptr` ist für Container geeignet. Der Sinn von `weak_ptr`-Objekten ist es, zyklische Datenstrukturen unterbrechen zu können. Wenn in einer zyklischen Datenstruktur ein Knoten auf den nächsten mit einem `shared_ptr` verweist, kann der Benutzungszähler bei keinem Knoten null werden, d.h. der Destruktor wird nicht aufgerufen. Ein einfaches Beispiel:

```
// cppbuch/k33/weakptr/main.cpp
#include <iostream>
#include <memory>
using namespace std;

struct ZyklStruktur { // Konstruktor für das Beispiel nicht erforderlich
   ~ZyklStruktur() {
      cout << "Destruktor ~ZyklStruktur() aufgerufen" << endl;
   }
   weak_ptr<ZyklStruktur> nachbar; // *** siehe Text
};

void f() {
   ZyklStruktur* a1 = new ZyklStruktur;
   ZyklStruktur* a2 = new ZyklStruktur;
   // zyklische Struktur (gegenseitige Verweise) herstellen:
   a1->nachbar = shared_ptr<ZyklStruktur>(a2);
   a2->nachbar = shared_ptr<ZyklStruktur>(a1);
}

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

Der Destruktor wird für die Objekte `*a1` und `*a2` aufgerufen – es gibt kein Problem. Wenn aber die ***-markierte Zeile durch `shared_ptr<ZyklStruktur> nachbar;` ersetzt würde, blieben sie, wenn `main()` nicht beendet würde, unerreichbar auf dem Heap! `weak_ptr` besitzt die folgenden Methoden:

- `long use_count()` gibt `sptr.use_count()` zurück, wobei `sptr` das bei der Konstruktion verwendete `shared_ptr`-Objekt ist. Falls `sptr == NULL` ist, wird 0 zurückgegeben.
- `bool expired()` gibt `use_count() == 0` zurück.
- `shared_ptr<T> lock()` gibt das zugeordnete `shared_ptr`-Objekt zurück, falls vorhanden, andernfalls `shared_ptr()`.

33.2 new mit vorgegebenem Speicherort

Der Header <new> enthält die Operatoren new, new[], delete und delete[], die in Abschnitt 5.4 ab Seite 200 beschrieben werden. Die Darstellung des Funktionszeigers new_handler und der zugehörigen Funktion set_new_handler zur Fehlerbehandlung sowie die Klasse bad_alloc finden sich in Abschnitt 8.2 ab Seite 312. Hier wird deshalb nur eine im Header <new> vorhandene weitere Form des new-Operators beschrieben, die »Placement-Form«.

```
// Placement-Form für new
void* operator new (std::size_t size, void *ptr);
void* operator new[](std::size_t size, void *ptr);
```

Zurückgegeben wird ptr. Die Placement-Operatoren dürfen nicht durch eigene mit derselben Signatur ersetzt werden. Die Placement-Form ist nützlich, wenn die Adresse, an der ein Objekt abgelegt werden soll, schon vorher bekannt ist. Bei vielen Objekten kann dies durchaus Zeit sparen, weil das normale new erst das Betriebssystem um Speicher ersucht.

Die entsprechenden Placement-delete-Operatoren gibt es nur der Form halber. Sie bewirken nichts, weil durch ein Placement-new kein neuer Speicher zugewiesen wird, sondern nur Objekte in einem vorhandenen Speicher angelegt werden. Es muss also kein Speicher freigegeben werden. Wenn Sie allerdings das Placement-new für eine Klasse überladen, sollten Sie auch das zugehörige Placement-delete schreiben. Zur Begründung siehe [ScM]. Ein Beispiel für das Placement-new:

```
// cppbuch/k33/placement/placement.cpp
#include<iostream>
#include<new>
using namespace std;

class Irgendwas {
  public:
   Irgendwas() : id(++maxid) { }
   void machwas() const {
     cout << "Id = " << id << endl;
   }
  private:
   int id;
   static int maxid;
};

int Irgendwas::maxid = 0;

int main() {
  char vielPlatz[1000*sizeof(Irgendwas)] = {0}; // mit 0 initialisieren

  // Ein Objekt in vielPlatz anlegen:
  Irgendwas* p = new (vielPlatz) Irgendwas; // Objekt 1
  p->machwas();
```

```
  // Weitere 10 Objekte mit Array-Operator anlegen:
  char* naechsteAdresse = (char*)p + sizeof(Irgendwas);
  new (naechsteAdresse) Irgendwas[10]; // Objekte 2 bis 11

  // Alle 11 Objekte abfragen (1-11)
  for(int i = 0; i < 11; ++i) {
     cout << i << ": ";
     p++->machwas();
  }

  p->machwas(); // Ausgabe 0!
}
```

Die nächste Position ist nicht belegt, das heißt, der letzte Aufruf `p->machwas()` gibt 0 aus, weil das Feld `vielPlatz` mit 0 initialisiert wurde. `p` zeigt auf einen Bereich, in dem gar kein Objekt des Typs `Irgendwas` angelegt wurde. Nichtsdestoweniger wird der Zeiger `p` so interpretiert, als zeige er auf ein Objekt. Die Daten sind aber alle 0.

33.3 Hilfsfunktionen

Im Header `<memory>` sind unter anderem die folgenden Klassen und Funktionen vertreten:

- `template<typename T> class allocator`
  Die `allocator`-Klasse stellt die Dienstleistungen bereit, die zur Beschaffung von Speicherplatz notwendig sind. In Abhängigkeit vom Memory-Modell wird ein passender Allokator vom System bereitgestellt, sodass sich ein Anwender nicht darum kümmern muss, es sei denn, er möchte selbst spezielle Memory-Funktionen realisieren, zum Beispiel eine Speicherverwaltung mit garbage collection. Solche Aufgabenstellungen sind recht speziell, sodass auf eine Beschreibung hier verzichtet wird. Wer einen Allokator selbst schreiben möchte, sei auf [Alex] verwiesen.
- `template<class OutputIterator, class T>`
  `class raw_storage_iterator`
  Die Klasse ermöglicht es, Daten in nicht-initialisierten Speicher zu schreiben. Es wird auf die Beschreibung in [ISOC++] verwiesen.
- `template<typename T>`
  `pair<T*, ptrdiff_t> get_temporary_buffer(ptrdiff_t n)`
  Diese Funktion beschafft temporären Speicher. Zurückgegeben wird ein Paar, das die Speicheradresse und den verfügbaren Platz in Einheiten von `sizeof(T)` enthält, das heißt, `n` bei Erfolg und 0, falls die Speicherbeschaffung nicht gelingt.
- `template<typename T> void return_temporary_buffer(T* p)`
  Diese Funktion gibt einen mit `get_temporary_buffer()` an der Stelle `p` beschafften Speicher wieder frei.

- `template<class InputIterator, class ForwardIterator>`
  `ForwardIterator uninitialized_copy(InputIterator first,`
  `InputIterator last, ForwardIterator result)`
  Diese Funktion kopiert alle Werte des Bereichs `[first, last)` nach `result`. Beispiel:

```
// cppbuch/k33/placement/uninitcopy.cpp
#include<algorithm>
#include<iostream>
#include<new>
#include<vector>
#include<showSequence.h>
using namespace std;

int main() {
   vector<int> v1(10), v2(10);
   fill(v1.begin(), v1.end(), 999); // v1 mit Daten füllen
   // Jetzt v1 nach v2 kopieren:
   uninitialized_copy(v1.begin(), v1.end(),v2.begin());
   showSequence(v2);
}
```

  Voraussetzung ist, dass der `operator*()` eines Iterators ein Objekt zurückgibt, dessen Adressoperator definiert ist und der einen Zeiger auf das Objekt zurückgibt. Das ist bei einem `vector<T>` der Fall, sofern `T` nicht `bool` ist. An diese Stelle wird der Speicherinhalt mit dem von oben bekannten Placement-new kopiert. Die Wirkung ist

```
for (; first != last; ++result, ++first) {
    new (static_cast<void*>(&*result))          // Ziel-Speicherplatz
        // abzulegendes Objekt:
        typename iterator_traits<ForwardIterator>::value_type(*first);
}
```

- `template<class ForwardIterator, class T>`
  `void uninitialized_fill(ForwardIterator first, ForwardIterator last, const T& x)`
  Diese Funktion füllt alle Positionen im Bereich `[first, last)` mit Kopien von `x`. Beispiel mit dem Vektor `V1` von oben:

```
int Wert = 17;
uninitialized_fill(v1.begin(), v1.end(), Wert);
```

- `template<class ForwardIterator, class Size, class T>`
  `void uninitialized_fill_n(ForwardIterator first, Size n,`
  `const T& x)`
  Diese Funktion füllt `n` Positionen ab Position `first` mit Kopien von `x`. Im Beispiel mit dem Vektor `v1` und dem Wert von oben werden 20 Werte eingetragen:

```
uninitialized_fill_n(v1.begin(), 20, Wert);
```