Einführung in die Programmierung mit C++

Dynamische Speicherverwaltung in Klassen[-hierarchien]

Uwe Naumann

Informatik 12:
Software and Tools for Computational Engineering (STCE)

RWTH Aachen

Inhalt

Outline

Dynamische Speicherverwaltung in Klassen

Allozierung und Initialisierung

- Der Zugriff auf dynamischen Speichern innerhalb von Klassen erfolgt über Zeiger, z.B. v.
- Der zu verwaltende dynamische Speicher sollte bei Konstruktion eines Objekts alloziert werden, z.B. innerhalb der Initialisierungsliste von Konstruktoren.

```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public:
5    vector(size_t n) : v(new T[n]), n(n) {
6      for (size_t i=0;i<n;i++) v[i]=42;
7    }
8  // ...
9  };
```

- Allozierung von dynamischem impliziert den Aufruf des Standardkonstruktors (in Feldern für jedes Element), was typischerweise die Initialisierung zur Folge hat.
- Alternative Initialisierung sollte innerhalb des Konstruktors vorgenommen werden.

Outline

Dynamische Speicherverwaltung in Klassen

Deallozierung

- Der bei Konstruktion von Objekten allozierte dynamische Speicher wird bei Deallozierung der Objekte nicht dealloziert. Es wird lediglich der für den Zeiger benötigte (statische) Speicher freigegeben, was potentiell *memory leaks* zur Folge hat.

```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public:
5    vector(size_t n) : v(new T[n]), n(n) {
6      for (size_t i=0;i<n;i++) v[i]=42;
7    }
8    ~vector() { delete [] v; }
9  // ...
10 };
```

- Bei Deallozierung eines Objekts wird dessen Destruktor ausgeführt.
- Im Destruktor sollte der durch das Objekt belegte dynamische Speicher dealloziert werden.
- Allozierung und Deallozierung von dynamischem Speicher sind nicht auf Konstruktor und Destruktor beschränkt, was jedoch die Gefahr von *memory leaks* erhöht.

AUFWACHEN!

Was ist an folgendem C++ Programm problematisch?

```
1  template<typename T>
2  class A {
3    T* d=nullptr;
4  public:
5    A() : d(new T) {}
6  };
7
8  int main(int argc, char*[]) {
9    A<int>* a;
10   if (argc>1) a=new A<int>;
11   delete a;
12   return 0;
13 }
```

WACH BLEIBEN!

OK, Deallozierung hat gefehlt, aber ...

... was bleibt problematisch?

```
1  template<typename T>
2  class A {
3    T* d=nullptr;
4  public:
5    A() : d(new T) {}
6    ~A() { delete d; }
7  };
8
9  int main(int argc, char*[]) {
10   A<int>* a;
11   if (argc>1) a=new A<int>;
12   delete a;
13   return 0;
14 }
```

Wach?

Bei Aufruf ohne Kommandozeilenparameter kann es zu einem *segmentation fault* aufgrund der versuchten Deallozierung eines ungültigen Zeigers kommen. Initialisierung mit `nullptr` schafft Abhilfe.

```
1  template<typename T>
2  class A {
3    T* d=nullptr;
4  public:
5    A() : d(new T) {}
6    ~A() { delete d; }
7  };
8
9  int main(int argc, char*[]) {
10   A<int>* a=nullptr;
11   if (argc>1) a=new A<int>;
12   delete a;
13   return 0;
14 }
```

Outline

Komposition vs. Aggregation

Verantwortlichkeiten bei Dynamischer Speicherverwaltung

Ein Objekt A kann ein anderes Objekt B enthalten. Beide können sowohl im statischen als auch im dynamischen Speicher alloziert sein.

```
1  #include "vector.hpp"
2
3  template<typename T>
4  class composition {
5    vector<T> *v; // owned
6  public:
7    composition() : v(new vector<T>) {}
8    composition(vector<T> *v) : v(v) {}
9    ~composition() { delete v; }
10 };
11
12 template<typename T>
13 class aggregation {
14   vector<T> *v; // referenced
15 public:
16   aggregation(vector<T> *v) : v(v) {}
17   ~aggregation() {}
18 };
```

- Ist die "Lebenszeit" von B an die von A gekoppelt, d.h. Deallozierung von A impliziert die Deallozierung von B, so liegt eine Komposition vor.
- Ist die "Lebenszeit" von B nicht an die von A gekoppelt, d.h. B kann existieren obwohl A bereits deallozierd wurde, so liegt eine Aggregation vor.
- Diese Unterscheidung ist essentiell für die Vermeidung von *memory leaks*.

AUFWACHEN!

Was ist an folgendem C++ Programm problematisch?

```
1  struct B {};
2
3  struct A {
4    B* b=nullptr;
5    ~A() { delete b; }
6  };
7
8  int main(int argc, char*[]) {
9    B b; A a; a.b=&b;
10   if (argc>1) a.b=new B;
11   return 0;
12 }
```

Wach?

Aufruf ohne Kommandozeilenargument führt zu Fehler aufgrund versuchter expliziter Deallozierung eines statischen Speicherbereichs. Besser ...

```
1  struct B {};
2
3  class A {
4    bool heap=false;
5    B* b=nullptr;
6  public:
7    A(B &b) : b(&b) {}
8    A() : heap(true) { b=new B; }
9    ~A() { if (heap) delete b; }
10 };
11
12 int main() {
13   B b; A a1,a2(b);
14   return 0;
15 }
```

... Inspektion mit gdb.

Outline

Kopieren

Problem


```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public:
5    vector(size_t n) : v(new T[n]), n(n) {
6      for (size_t i=0;i<n;i++) v[i]=42;
7    }
8    ~vector() { delete [] v; }
9  // ...
10 };
11
12 int main(int argc, char*[]) {
13   vector<int> v(argc), v_copy=v;
14   // do something useful
15   return 0;
16 }
```

- Beim Kopieren wird automatisch nur der belegte statische Speicher, nicht aber der belegte dynamische Speicher, dupliziert. Es wird demnach automatisch nur der Zeiger, nicht aber der durch ihn referenzierte dynamische Speicher, dupliziert.
- Das kann gewollt sein. Typischerweise ist es das aber nicht, z.B. führt nebenstehendes Programm zu einem Laufzeitfehler infolge versuchter doppelter Deallozierung:

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

Kopieren

Kopierkonstruktor

Bei Verwaltung von dynamischem Speicher durch Objekte eines nutzuerdefinierten Typs sollte ein spezieller Kopierkonstruktor zur Verfügung gestellt werden. Dieser sollte sich um die Duplizierung des dynamischen Speichers und um dessen korrekte Referenzierung innerhalb der Kopie kümmern.

```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public:
5    vector(size_t n) : v(new T[n]), n(n) {
6      for (size_t i=0;i<n;i++) v[i]=42;
7    }
8    ~vector() { delete [] v; }
9    vector(const vector& x) : v(new T[x.n]), n(x.n) {} // copy constructor
10 // ...
11 };
```

Kopierkonstruktoren erwarten konstante Referenzen auf das zu kopierende Objekt als einzigen Parameter und verhalten sich sonst wie all anderen Konstruktoren.

Neben expliziten Aufrufen bzw. äquivalenten Initialisierungsausdrücken werden Kopierkonstruktoren automatisch in folgenden Situationen aufgerufen:

```
1  #include <iostream>
2
3  struct A {
4    A(){}
5    A(const A&) {
6      std::cout << "copy" << std::endl;
7    }
8  };
9
10 A f(A a) { return a; }
11
12 int main() {
13   A v, v_copy1(v), v_copy2=v;
14   v_copy1=f(v);
15   return 0;
16 }
```

- Bei Parameterübergabe an Funktionen "by value"
- bei Rückgabe von Resultaten von Funktionen

Z.B. wird durch das nebenstehende Programm

```
copy
copy
copy
copy
```

ausgegeben.

AUFWACHEN!

Kann folgendes C++ Programm fehlerfrei übersetzt werden?

```
1  class A {
2    int *p=nullptr;
3  public:
4    A() : p(new int) {}
5    ~A() { delete p; }
6    A(A a) : p(new int) { *p=*(a.p); }
7  };
8
9  int main() { A a,b=a; }
```

Wach?

Nein (rekursiver Aufruf des Kopierkonstruktors), aber folgendes schon:

```
1  class A {
2    int *p=nullptr;
3  public:
4    A() : p(new int) {}
5    ~A() { delete p; }
6    A(A &a) : p(new int) { *p=*(a.p); }
7  };
8
9  int main() { A a,b=a; }
```

Beachte: Argument des Kopierkonstruktors sollte **const** sein (muss es aber nicht).

Outline

Zuweisen

Problem


```
1  template<typename T>
2  class vector {
3  protected:
4    T* v; size_t n;
5  public:
6    vector(size_t n) : v(new T[n]), n(n) {
7      for (size_t i=0;i<n;i++) v[i]=42;
8    }
9    ~vector() { delete [] v; }
10 };
11
12 int main(int argc, char*[]) {
13   vector<int> v(argc), v_copy(argc);
14   v_copy=v;
15   // do something useful
16   return 0;
17 }
```

- Analog zum Kopieren wird bei Zuweisung automatisch nur der belegte statische Speicher, nicht aber der belegte dynamische Speicher, dupliziert. Es wird demnach automatisch nur der Zeiger, nicht aber der durch ihn referenzierte dynamische Speicher, dupliziert.
- Das kann wiederum gewollt sein. Typischerweise ist es das aber nicht; z.B. generiert nebenstehendes Programm eine *memory leak* und führt zu einem Laufzeitfehler infolge versuchter doppelter Deallozierung:

```
free(): double free detected in tcache 2
Aborted (core dumped)
```

Zuweisen

Zuweisungsoperator (Variante 1)

Analog zum Kopierkonstruktor sollte bei Verwaltung von dynamischem Speicher durch Objekte eines nutzerdefinierten Typs ein spezieller Zuweisungsoperator als Objektfunktion der linken Seite der Zuweisung zur Verfügung gestellt werden.

```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public: // ...
5    vector& operator=(const vector& x) { // assignment operator
6      assert(n==x.n);
7      for (size_t i=0;i<n;i++) v[i]=x.v[i];
8      return *this;
9    }
10 // ...
11 };
```

Zuweisungsoperatoren sollten eine konstante Referenz auf das Objekt auf der rechten Seite der Zuweisung als einzigen Parameter übergeben bekommen. Ihr Rückgabewert ist eine Referenz auf das Objekt of the linken Seite der Zuweisung (*this).

Zuweisungsoperator (Variante 2)


```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public:
5  // ...
6    vector& operator=(const vector& x) {
7      if (n==x.n)
8        for (size_t i=0;i<n;i++) v[i]=x.v[i];
9      else {
10       delete [] v; n=x.n; v=new T[n];
11       for (size_t i=0;i<n;i++) v[i]=x.v[i];
12     }
13     return *this;
14   }
15 // ...
16 };
```

- Variante 1 funktioniert nur für Vektoren gleicher Größe.
- Im Allgemeinen würde Zuweisung eines Vektors der Länge n_1 zu einem Vektor der Länge n_2 die Länge des Vektors auf der linken Seite der Zuweisung auch auf n_1 setzen und eine entsprechende Kopie des dynamischen Speichers anlegen müssen.
- Ein Reallozierung von Speicher müsste durchgeführt werden.

Zuweisen

Zuweisungsoperator (Variante 3)


```
1  template<typename T>
2  class vector {
3    T* v; size_t n;
4  public:
5  // ...
6    vector& operator=(const vector& x) {
7      if (&x==this) return *this;
8      if (n==x.n)
9        for (size_t i=0;i<n;i++) v[i]=x.v[i];
10     else {
11       delete [] v; n=x.n; v=new T[n];
12       for (size_t i=0;i<n;i++) v[i]=x.v[i];
13     }
14     return *this;
15   }
16 // ...
17 };
```

- Variante 2 resultiert in einem undefinierten Zustand des Vektors bei Selbstzuweisung der Form x=x; dieser Fall sollte speziell behandelt werden indem dann im Prinzip gar nichts passiert → Zeile 7.
- Selbstzuweisungen müssen nicht offensichtlich sein, z.B.

```
int i=42, ir=i, ip=&ir;
ir=*ip; // self-assignment
```

wobei die Belegung insbesondere von Zeigern erst zur Laufzeit des Programms feststehen kann.

AUFWACHEN!

Welchen Wert hat *(b.p) kurz vor Beenden des Programms?

```
1  class A {
2    int *p=nullptr;
3  public:
4    A(int i) : p(new int) { *p=i; }
5    ~A() { delete p; }
6    A(const A &a) : p(new int) {}
7    A& operator=(A a) {
8      if (&a!=this) *p=*(a.p);
9      return *this;
10   }
11 };
12
13 int main() {
14   A a(42),b(24); b=a;
15   return 0;
16 }
```

Wach?

```
1  class A {
2    int *p=nullptr;
3  public:
4    A(int i) : p(new int) { *p=i; }
5    ~A() { delete p; }
6    A(const A &a) : p(new int) {}
7    A& operator=(A a) {
8      if (&a!=this) *p=*(a.p);
9      return *this;
10   }
11 };
12
13 int main() {
14   A a(42),b(24); b=a;
15   return 0;
16 }
```

0

WACH BLEIBEN!

Welchen Wert hat *(b.p) kurz vor Beenden des Programms?

```
1  class A {
2    int *p=nullptr;
3  public:
4    A(int i) : p(new int) { *p=i; }
5    ~A() { delete p; }
6    A(const A &a) : p(new int) {}
7    A& operator=(A &a) {
8      if (&a!=this) *p=*(a.p);
9      return *this;
10   }
11 };
12
13 int main() {
14   A a(42),b(24); b=a;
15   return 0;
16 }
```

Noch Wach?

```
1  class A {
2    int *p=nullptr;
3  public:
4    A(int i) : p(new int) { *p=i; }
5    ~A() { delete p; }
6    A(const A &a) : p(new int) {}
7    A& operator=(A a) {
8      if (&a!=this) *p=*(a.p);
9      return *this;
10   }
11 };
12
13 int main() {
14   A a(42),b(24); b=a;
15   return 0;
16 }
```

42

Beachte: Argument des Kopierkonstruktors sollte **const** sein (muss es aber nicht).

- *Rule of Zero*

  Vermeide eigenen Destruktor, Kopierkonstruktor, Zuweisungsoperator falls möglich.

- *Rule of Three*

  Implementiere eigenen Destruktor und (!) Kopierkonstruktor und (!) Zuweisungsoperator falls nötig.

- *Rule of Five* ... ab C++11 in Verbindung mit move-Semantik → Advanced C++

Outline

Polymorphismus

Dynamische Speicherveraltung in Klassenhierarchien


```
1  template<typename T>
2  struct abstract_array {
3    virtual ~abstract_array() {}
4    virtual T& operator[](int)=0;
5  };
6
7  template<typename T>
8  class abstract_vector : public abstract_array<T> {
9  protected:
10   T* v;
11 public:
12   abstract_vector(int n) : v(new T[n]) {}
13   ~abstract_vector() { delete [] v; }
14 };
15
16 template<typename T>
17 class unsafe_vector : public abstract_vector<T> {
18   using abstract_vector<T>::v;
19 public:
20   unsafe_vector(int n) : abstract_vector<T>(n) { }
21   T& operator[](int i) { return v[i]; }
22 };
```

- Polymorphism erlaubt das Referenzieren von Objekten eines abgeleiteten Typs (z.B. unsafe_vector) via Zeiger bzw. Referenzen auf einen Basistyp (z.B. abstract_array).
- Destruktoren in (abstrakten) Basisklassen sollten immer **virtual** sein, um die korrekte Deallozierung von in abgeleiteten Klassen verwaltetem dynamischen Speicher zu garantieren.

Experimente mit folgendem Beispielprogramm mit und ohne **virtual** an den entsprechenden Stellen von vector_hierarchy.hpp verdeutlichen die skizzierten Effekte.

```
1  #include "vector_hierarchy.hpp"
2
3  int main() {
4    const int n=3;
5    abstract_array<int> *v=new unsafe_vector<int>(n);
6    // do something useful
7    delete v;
8    return 0;
9  }
```

Live: Diskussion der gesamten Fallstudie

Polymorphismus

Virtueller **operator=**

```
1  template<typename T>
2  class vector {
3  protected:
4    T* v; size_t n;
5  public:
6    // ...
7    virtual vector& operator=(const vector& x)
         ;
8    // ...
9  };
10
11 template<typename T>
12 class reverse_vector : public vector<T> {
13   // ...
14 public:
15   reverse_vector(size_t n) : vector<T>(n) {}
16   /// reverses order of the entries
17   reverse_vector<T>& operator=(const
         vector<T>& x);
18 };
```

- In einer weiteren Fallstudie ist die Polymorphie durch einen virtuellen Zuweisungsoperator in der Basisklasse implementiert.
- Der Rückgabetyp ist nicht Teil der Signatur. Damit führen Zuweisungen von Objekten von Typ vector zu Referenzen oder Zeigern vom Typ vector, welche mit Objekten vom Typ reverse_vector assoziiert sind, zur Umkehrung der Reihenfolge der Vektorelemente.

```
1  #include "vector.hpp"
2  #include "reverse_vector.hpp"
3
4  #include <iostream>
5
6  int main(int argc, char*[]) {
7    vector<int> x(argc);
8    for (size_t i=0;i<x.size();i++) x[i]=i;
9    std::cout << x << std::endl;
10   reverse_vector<int> y(1);
11   y=x;
12   std::cout << y << std::endl;
13   vector<int>& yr=y;
14   yr=x;
15   std::cout << yr << std::endl;
16   return 0;
17 }
```

Aufruf des Programms mit 3 Kommandozeilenargumenten resultiert in folgender Ausgabe:

```
[ 0 1 2 3 ]
[ 3 2 1 0 ]
[ 3 2 1 0 ]
```

Für den Fall, dass der Zuweisungsoperator in vector nicht virtuell ist, wird

```
[ 0 1 2 3 ]
[ 3 2 1 0 ]
[ 0 1 2 3 ]
```

ausgegeben.

Outline

Wäre es nicht schön, wenn man manuell dynamischen Speicher allozieren, dessen Deallozierung jedoch automatisieren könnte?

```
1  #include<iostream>
2  #include<cassert>
3
4  void f(int i) {
5    int* p=new int(i);
6    std::cout << *p << " " << p << std::endl;
7    if (i) f(i-1);
8    // p runs out of scope; *p not deallocated -> memory leaks
9  }
10
11 int main(int c, char* v[]) {
12   assert(c==2);
13   f(std::stoi(v[1]));
14   return 0;
15 }
```

Die Standardbibliothek bietet spezielle *smart pointer*-Typen an.

Schlaue Zeiger

Nutzerdefiniert


```
1  #include<iostream>
2
3  template<typename T>
4  struct smart_ptr {
5    T* p;
6    smart_ptr(T* p) : p(p) {};
7    ~smart_ptr() {
8      std::cout << "~" << std::endl;
9      delete p;
10   }
11 };
12
13 void f(int i) {
14   smart_ptr<int> p(new int(i));
15   std::cout << *(p.p) << std::endl;
16 }
17
18 int main() {
19   f(42);
20   return 0;
21 }
```

- Ein Zeiger `p` auf Daten generischen Typs wird in der Klasse `smart_ptr` definiert.
- Dieser wird bei Allozierung eines Objekts der Klasse `smart_ptr` mit der Basisadresse eines dynamischen Speicherbereichs initialisiert.
- Der referenzierte dynamische Speicher wird bei Verlassen des Gültigkeitsbereichs von Objekten vom Typ `smart_ptr` automatisch dealloziert.
- Der Gültigkeitsbereich von `p` ist hier gleich `f`; folgende Ausgabe wird erzeugt:

```
42
~
```

Schlaue Zeiger

Dynamische 1D-Felder


```
1  #include<iostream>
2
3  struct A {
4    ~A() {
5      std::cout << "~" << std::endl;
6    }
7  };
8
9  template<typename T>
10 struct smart_array {
11   T* p;
12   smart_array(T* p) : p(p) {};
13   ~smart_array() {
14     delete [] p;
15   }
16 };
17
18 void f(int n) {
19   smart_array<A> p(new A[n]);
20 }
21
22 int main() { f(3); return 0; }
```

- Schlaue dynamische Felder können analog implementiert werden; hier: dynamische Allozierung eines 1D-Feldes
- Beachte:
  - Fehler bei Deallozierung mittels **delete** p; Laufzeitfehler
  - Korrekte Deallozierung mittels **delete** [] p

```
1  #include<iostream>
2  #include<memory>
3
4  struct A {
5    ~A() {
6      std::cout << "~" << std::endl;
7    }
8  };
9
10 void f(int n) {
11   std::unique_ptr<A[]> p(new A[n]);
12   for (int i=0;i<n;i++)
13     std::cout << &p[i] << std::endl;
14 }
15
16 int main() {
17   f(3);
18   return 0;
19 }
```

- Die Standardbibliothek stellt z.B. schlauer exklusive Zeiger std::unique_ptr auf Elemente generischen Typs zur Verfügung.
- Diese sind zum Zeitpunkt der Allozierung zu initialisieren.
- Die Deallozierung des belegten dynamischen Speichers erfolgt bei Verlassen des Gültigkeitsbereichs (hier f)
- Entsprechende dynamische Felder von Elementen des Typs A erhält man mittels durch die Typspezifikation A[].

Zusammenfassung

Allozierung und Initialisierung

Deallozierung

Komposition vs. Aggregation

Kopieren

Zuweisen

Polymorphismus

Schlaue Zeiger