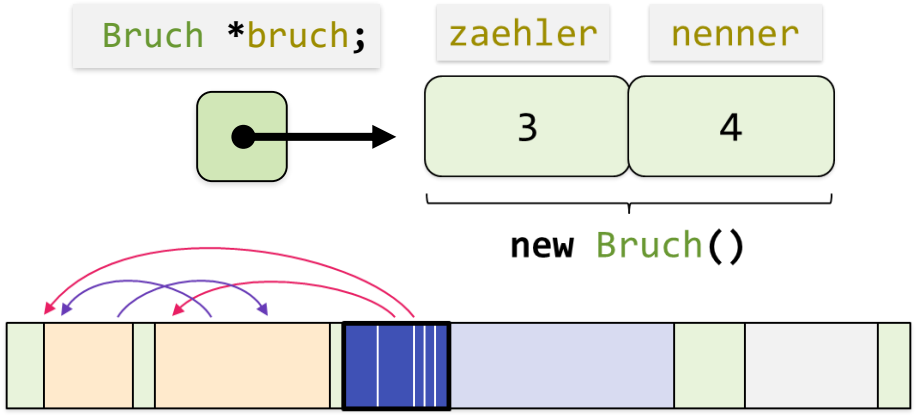
Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020



Foliensatz #06

Generische Programmierung

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Low-Level ist gefährlich

Wir haben bisher gesehen:

- C++ kann nah an Maschinenebene arbeiten
- Schnell, effizient, sehr kontrollierbar
- Aber Gefahr bez. Speichersicherheit
  - Speicherlecks
  - Ungültige Zeiger
    - z.B. Bereichsüberschreitungen („Buffer Overrun“)
    - z.B. doppelte Freigabe, Zugriff nach Freigabe (delete/free)
    - z.B. Verweise auf bereits abgebaute Stack-Variablen

Lösung: Kapselung in Bibliotheken

Sichereres C++

Schnell + Sicher?

- Beschränke kritischen Code auf wenige Bausteine
- Analysiere & teste diese gut
- Danach Arbeit mit abstrakteren, weniger fehleranfälligen „higher-level“ Komponenten

Stichwort: „Efficient Low-Level Abstractions“

- Wir schauen uns das ganze an einem Beispiel an
  - Container für Daten: Dynamisches Array
  - Alternative zu „DatenTyp* somePtr“
  - Mehr Komfort, Checks möglich

Containerklassen

Klasse für Listen (Arrays)

Containerklasse: Listen

```
struct IntList {
    int *memory;
    int size;
    // Konstruktur
    IntList(unsigned initialSize = 0) {
        memory = new int[initialSize]; // Speicher reservieren (0 erlaubt)
        size = initialSize;
    }
    // Neu: Destruktor (bei Löschen des Objektes)
    ~IntList(unsigned initialSize = 0) {
        delete[] memory; // Speicher freigeben
    }
    // Zugriff auf Elemente
    int &operator[](unsigned index) {
        if (index >= size) {throw std::out_of_range("out of bounds");}
        return memory[index];   // ↑↑ Neu: Exceptions – ähnlich wie in Python
    }
};
```

Klasse für Listen (Arrays)

Containerklasse: Listen

```
struct IntList {
    …
    // copy-Konstruktur
    IntList(const IntList &other) {
        memory = new int[other.size]; // Speicher reservieren
        size = other.size;            // Daten kopieren
        for (int i=0; i<size; i++) {  // Daten kopieren
            memory[i] = other.memory[i];
        }
    }
    // Zuweisungsoperator
    void operator=(const IntList &other) {
        delete[] memory;
        memory = new int[other.size]; // Speicher reservieren
        size = other.size;            // Daten kopieren
        for (int i=0; i<size; i++) {  // Daten kopieren
            memory[i] = other.memory[i];
        }
    }
}
```

Benutzung

```
IntList *doSomething(int num) {
   IntList l1(3);
   for (int i=0; i<42; i++) {
      IntList l2(3); // Konstrukturaufruf l2! (42 mal!)
      l2[0] = 23;
      l2[1] = 42;
      l2[2] = 1337;
   } // Destrukturaufruf l2! (42 mal!)
   IntList *l3 = new IntList(3); // Konstrukturaufruf l3
   return l3; // l3 überlebt return, da auf Heap angelegt
} // Destrukturaufruf l1! (einmal pro Aufruf von doSomething)

void main() {
   IntList *my_l = doSomething();
   my_l[2] = 1337;
   delete my_l;
}
```

RAII – Halbautomatischer Speicher

```
{
   IntList l2(3); // Konstruktur
   l2[0] = 23;
   l2[1] = 42;
   l2[2] = 1337;
} // Destruktur (automatisch)
```

„RAII“-Stil

```
{
   IntList *l2 = new IntList(3);
   l2[0] = 23;
   l2[1] = 42;
   l2[2] = 1337;
}
…
delete l2;
```

Kein „RAII“-Stil
(üblich auch in JAVA, dort ohne delete)

RAII

- „Resource Acquisition Is Initialization“
- Lokale Variablen benutzen, um Ressource (z.B. Speicher) automatisch freizugeben
- **Tipp:** Nutzen, wann immer möglich (geht nicht immer)

Speicherverwaltungsstrategien

Wir brauchen ein Architekturmodell

- RAII
  - Einfach und sicher, aber Lebensdauer beschränkt auf Blöcke
  - Keine „Persistenz“ (z.B. Dokument einer Anwendung)
- „Ownership“-Modell
  - Baumstrukturierter Objektgraph
  - Eltern-Objekt ist „Owner“ der Kinder (verantwortlich für delete)
  - Querverweise (non-owned) möglich, kein delete
- Referenz-Counting
  - Für azyklisch-gerichtete Graphen (mehrere Owner)
- Allgemeine Graphen: Sonderlösungen oder GC

Generische Container

Container für verschiedene Typen

Containerklasse: Listen

```
template <typename T>
struct List {
    T *memory;
    int size;
    // Konstruktur
    List(unsigned initialSize = 0) {
        memory = new T[initialSize]; // Speicher reservieren (0 erlaubt)
        size = initialSize;
    }
    // Destruktor (bei Löschen des Objektes)
    ~List(unsigned initialSize = 0) {…}
    // Zugriff auf Elemente
    T &operator[](unsigned index) {…}
    // Copy-Constructor
    List(const List<T> &other) {…}
    // Zuweisungsoperator
    void operator=(const List<T> &other) {
};
```

Templates

```
List<int> l1(3);
l1[0] = 23;
l1[1] = 42;
l1[2] = 1337;
```

```
List<double> l2(3);
l2[0] = 23.0;
l2[1] = 42.0;
l2[2] = 1337.0;
```

```
List<Bruch> l3(1);
l3[0].zaehler = 23.0;
l3[0].nenner = 42.0;
```

Statische Typisierung

- Erfordert genaue Typspezifikation
- Viele doppelte Tipparbeit

Lösung: „templates“ (generische Programmierung)

- Typparamter für Klassen/Structs oder Funktionen
- Benutzung mit Spitzen Klammern (s.o.)

Generische Programmierung

Templates in C++

„Generische Programmierung“

- Eine Form von Polymorphie
- Der selbe Code kann unterschiedliche Dinge tun
- Andere Datentypen führen zu anderem Verhalten

Werkzeug in C++: Templates

- Präambel „**`template <…>`**“ erzeugt generischen
  - Typ (struct/class)
  - Member-Funktion (von structs/classes)
  - Funktion (alleinstehend)
- Benutzung mit **`Typ<…>`** bzw. **`Funktion<…>`**

Beispiele

```
// Generischer Typ
template <typename T>
struct List {
   T *memory;
   …
};

// Generische Funktion
template <typename T>
T square(T val) {
   return val*val;
}

// Member-Template
struct IntList {
   int *memory;
   template <typename T>
   void setElementAs(unsigned i, const T val)
      {memory[i] = (int)val;}
};
```

```
// Benutzung generischer Klassen
List<int> a;
b = List<int>();
```

```
// Benutzung generischer Funkt.
float a = square(2.0f); // implizit
double b = square<double>(2); // explizit
```

```
// Benutzung generischer Memberf.
IntList a;
a.setElementAs<double>(2.0);
a.setElementAs(2.0f); // implizit
```

Explizite Spezialisierung

```
// Beispiel für Member-Templates; funktioniert mit allen Templates
struct IntList {
   int *memory;

   …

   template <typename T>
   void setElementAs(unsigned i, const T val) {
      static_assert(false, "Type not permitted."); // requires C++11 or later
   }

   template <>
   void setElementAs<int>(unsigned i, const int val) {
      memory[i] = val;
   }
   template <>
   void setElementAs<double>(unsigned i, const double val) {
      memory[i] = (int)val;
      cout << "Warning: had to round value.";
   }
};
```

Template Parameter

```
// Generisches Dictionary (langsam mit Liste)
template <typename KeyType, typename ValueType>
struct Dictionary {

   struct Entry { // lokale Definition erlaubt (Name außerhalb ist Dictionary::Entry)
      ValueType val;
      KeyType key;
   };

   List<Entry> allEntries; // Speichert alle Einträge

   const ValueType &operator[](const KeyType &key) const {
      for (int i=0; i<allEntries.size(); i++) {
         if (allEntries[i].key == key)
            return allEntries[i].value;
      }
      throw SomeException(…);
   }

};
```

```
// read-only (Instanz nicht beschreiben)
// kann überladen werden! (r/w extra)
```

```
// Benutzung
Dictionary<string, int> a;
…
cout << a["Hello"];
```

Integer-Parameter

Containerklasse: Listen

```
template <typename T, unsigned size>
struct FixedList { // Speicher reservieren
    T memory[size];

    // Konstruktur nicht nötig
    // Speicher ist bereits reserviert!
    FixedList() {}
    // Destruktor auch nicht nötig
    ~FixedList() {}

    // Zugriff auf Elemente
    T &operator[](unsigned index) {…}
    // Copy-Constructor
    FixedList(const FixedList<T,size> &other) {…}
    // Zuweisungsoperator
    void operator=(const FixedList<T,size> &other) {…}
}
```

```
// Benutzung
FixedList<double, 3> threeDVector;

// Man kann auch Namen vergeben
// (geht mit allen Typen!)
typedef FixedList<float,3> Vector3f;
typedef FixedList<double,3> Vector3d;

Vector3f v = Vector3f();
```

Python FTW

Gibt es templates in Python?

- Nicht notwendig!
- Jedes Unterprogramm ist generisch!

```
def square(a):
    return a*a
```

  - Dynamisch typisiert
  - Jeder Objektyp kann benutzt werden
  - Nachteil: Prüfung erst bei Ablauf
    - Fehler werden spät gefunden
    - Weniger Information über Funktion
    - Langsam

Duck-Typing

„Duck Typing“

- Typen in C++ templates arbeiten wie Python Typen, nur zur Übersetzungszeit:
  - Bei Übersetzung der Instantiierung:
    - Konkrete Typen einsetzen
    - „Als wäre es so eingetippt worden“
    - Prüfen, ob der Code Sinn macht
      - Wenn ja: So übersetzen
      - Wenn nein: (Kryptischer) Fehler
- Python
  - Typen zur Laufzeit einsetzen
  - Code wird erfolgreich ausgeführt, wenn er Sinn macht