TECHNISCHE UNIVERSITÄT DORTMUND

Wintersemester 2007/08

Einführung in die Informatik für Naturwissenschaftler und Ingenieure

(alias **Einführung in die Programmierung**)

(Vorlesung)

Prof. Dr. Günter Rudolph
Fakultät für Informatik
Lehrstuhl für Algorithm Engineering

Inhalt

- Motivation: Hashen von beliebigen Objekten
- Einführung von Schablonen / Templates
- ADT Liste (… schon wieder: jetzt aber wirklich als ADT)
- ADT HashTable mit Schablonen

Beispiel

Aufgabe:

- Hashen von komplexen Zahlen

Was ist zu tun?

1. ADT Complex → OK
2. ADT ComplexList → Gähn …
3. ADT ComplexHashTable → Gähn …

} Immer die gleichen Operationen, nur mit anderen Typen/Klassen!

⇒ Wie beim „richtigen“ abstrakten Datentyp müsste der Typ / die Klasse der Anwendungsdaten ein **Parameter** der Klasse sein!

Auf die Schnelle … → ADT Complex

```
class Complex {
protected:
  int fRe, fIm;
public:
  Complex() : fRe(0), fIm(0) { }
  Complex(int aRe) : fRe(aRe), fIm(0) { }
  Complex(int aRe, int aIm) : fRe(aRe), fIm(aIm) { }

  int Re() { return fRe; }
  int Im() { return fIm; }

  void Print() { std::cout << fRe << "+" << fIm << "*I"; }

  bool operator== (const Complex& c) {
    return fRe == c.fRe && fIm == c.fIm;
  }
```

überladener Operator

Fortsetzung nächste Folie

Auf die Schnelle … → ADT Complex

Fortsetzung …

```
  Complex operator+ (const Complex& c) {
    return Complex(fRe + c.fRe, fIm + c.fIm);
  }
  Complex operator- (const Complex& c) {
    return Complex(fRe - c.fRe, fIm - c.fIm);
  }
  Complex operator- () {
    return Complex(-fRe, -fIm);
  }
  Complex operator* (const Complex& c) {
    int re = fRe * c.fRe - fIm * c.fIm;
    int im = fIm * c.fRe - fRe * c.fIm;
    return Complex(re, im);
  }
};
```

überladene Operatoren


```
class IntList {
private:
  int elem;
  IntList *head;
  IntList *next;

  bool Contains(IntList *aList, int aElem);
  IntList *Delete(IntList *aList, int aElem);
  void Delete(IntList *aList);
  void Print(IntList *aList);

public:
  IntList();
  IntList(int aElem);

  void Insert(int aElem);
  bool Contains(int aElem);
  void Delete(int aElem);
  void Print();

  ~IntList();
};
```

ADT ComplexList?

→ Wir analysieren erst einmal `class IntList`

`int` : Typ / Klasse der Nutzinformation

↓

ersetzen durch `Complex`

↓

ersetzen durch generischen Typ T

Wie drückt man so etwas in C++ sprachlich aus?

```
template <class T>
class List {

protected:
  T       elem;
  List<T> *head;
  List<T> *next;

  bool Contains(List<T> *aList, T& aElem);
  List<T> *Delete(List<T> *aList, T& *aElem);
  void Delete(List<T> *aList);
  void Print(List<T> *aList);

public:
  /* und so weiter */
};
```

Bedeutung:
Nachfolgende Konstruktion hat Klasse T als Parameter!

Nachfolgende Konstruktion ist **keine Klasse**, sondern Muster / Schablone einer Klasse.

Schablonen bzgl. Vererbung wie Klassen.

→ Echte Klassen werden **bei Bedarf** vom Compiler aus der Schablone erzeugt!

Was ändert sich bei der Implementierung?

Muss vor jeder Definition stehen!

Wird Name der Klasse.

```
template <class T> void List<T>::Insert(T& aElem) {
  if (Contains(aElem)) return;
  List<T> *newList = new List<T>(aElem);
  newList->next = head;
  head = newList;
}
```

Konstruktor

⇒ auf diese Weise muss der gesamte Code von `IntList` verallgemeinert werden!

Was ändert sich bei der Implementierung?

öffentliche Methode:

```
template <class T> void List<T>::Print() {
  Print(head);
}
```

private überladene Methode:

```
template <class T> void List<T>::Print(List<T> *aList) {
  static int cnt = 1;      // counter
  if (aList != 0) {
    Print(aList->elem); // war: cout << aList->elem;
    cout << (cnt++ % 4 == 0) ? "\n" : "\t";
    Print(aList->next);
  }
  else {
    cnt = 1;
    cout << "(end of list)" << endl;
  }
}
```

`Print()` ist überladene Hilfsfunktion → später

Was ändert sich bei der Implementierung?

öffentliche Methode:

```
template <class T>
bool List<T>::Contains(T& aElem) {
  return Contains(head, aElem);
}
```

private überladene Methode:

```
template <class T>
bool List<T>::Contains(List<T> *aList, T &aElem) {
  if (aList == 0) return false;
  if (Equal(aList->elem, aElem)) return true;
  return Contains(aList->next, aElem);
}
```

u.s.w.

`Equal(.,.)` ist überladene Hilfsfunktion! Alternative: Operator `==` überladen!

Anwendung

```
class IntList : public List<int> {
public:
  IntList() : List<int>() { }
};
```

```
class ComplexList : public List<Complex> {
public:
  ComplexList() : List<Complex>() { }
};
```

```
class StringList : public List<string> {
public:
  StringList() : List<string>() { }
};
```

Wie funktioniert das?
Der Compiler erzeugt aus obigen Angaben und zugehöriger Schablone **automatisch** die Klassendeklaration und –definition!

Was fehlt noch? → **Hilfsfunktionen `Print` und `Equal`**

```
void Print(int x) { cout << x; }
void Print(float x) { cout << x; }
void Print(string x) { cout << x; }

void Print(Complex x) { x.Print(); }
```

Operator << ist überladen!

Ebenso ==

```
bool Equal(int x, int y) { return x == y; }
bool Equal(float x, float y) { return x == y; }
bool Equal(string x, string y) { return x == y; }

bool Equal(Complex x, Complex y) { return x.Equal(y); }
```

Man könnte auch Operator == für **`Complex`** überladen!

⇒ Dann bräuchte Code in **`Contains`** etc. nicht geändert zu werden!

Erstes Fazit:

+ Code für Liste muss nur einmal formuliert werden: Wiederverwertbarkeit!
+ Mit dieser Technik kann man Listen für jeden Typ formulieren.
  - Man muss nur Konstruktor definieren und
  - ggf. die Hilfsfunktionen **`Print`** und **`Equal`** hinzufügen.
  - Anmerkung: copy-Konstruktor wäre auch sinnvoll. Warum?

– Verallgemeinerung des Codes kann mühsam werden.

– Operatoren müssen entweder überladen oder durch (überladene) Hilfsfunktionen ersetzt werden.

ADT ComplexHashTable?

```
class AbstractHashTable {
private:
  IntList **table;
protected:
  int maxBucket;
public:
  AbstractHashTable(int aMaxBucket);
  virtual int Hash(int aElem) = 0;
  bool Contains(int aElem);
  void Delete(int aElem);
  void Insert(int aElem);
  void Print();
  ~AbstractHashTable();
};
```

→ Wir analysieren erst einmal die Klasse **`AbstractHashTable`**

`int / IntList` : spezialisierte Klassen der Nutzinformation

↓

ersetzen durch **Schablonen**

↓

`int` → **`T&`**
`IntList` → **`List<T>`**

HashTable als Schablone: Deklarationsschablone

```
template <class T>
class HashTableTemplate {
private:
  List<T> **table;
protected:
  int maxBucket;
public:
  HashTableTemplate(int aMaxBucket);
  virtual int Hash(T& aElem) = 0;
  bool Contains(T& aElem);
  void Delete(T& aElem);
  void Insert(T& aElem);
  void Print();
  ~HashTableTemplate();
};
```


HashTable als Schablone: Definitionsschablone

Konstruktor

```
template <class T>
HashTableTemplate<T>::HashTableTemplate(int aMaxBucket) {
  maxBucket = aMaxBucket;
  table = new List<T> *[maxBucket];
  for (int i = 0; i < maxBucket; i++)
    table[i] = new List<T>();
}
```

Destruktor

```
template <class T>
HashTableTemplate<T>::~HashTableTemplate() {
  for (int i = 0; i < maxBucket; i++)
    delete table[i];
  delete[] table;
}
```

HashTable als Schablone: Definitionsschablone

```
template <class T>
void HashTableTemplate<T>::Insert(T& aElem) {
  table[Hash(aElem)]->Insert(aElem);
}
template <class T>
void HashTableTemplate<T>::Delete(T& aElem) {
  table[Hash(aElem)]->Delete(aElem);
}
template <class T>
bool HashTableTemplate<T>::Contains(T& aElem) {
  return table[Hash(aElem)]->Contains(aElem);
}
template <class T>
void HashTableTemplate<T>::Print() {
  for (int i = 0; i < maxBucket; i++)
    table[i]->Print();
}
```


Instantiierung der Schablone

```
class IntHashTable : public HashTableTemplate<int> {
public:
  IntHashTable(int aMaxBucket);
  int Hash(int& aElem);
};

class ComplexHashTable : public HashTableTemplate<Complex> {
public:
  ComplexHashTable(int aMaxBucket);
  int Hash(Complex& aElem);
};
```


Instantiierung der Schablone

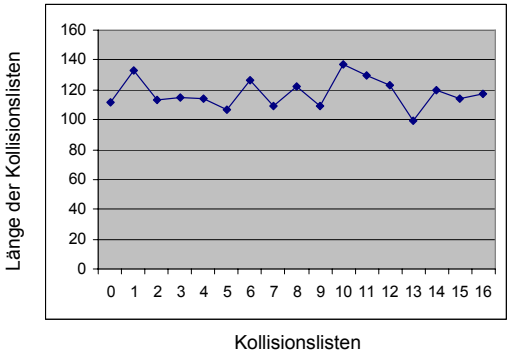
```
IntHashTable::IntHashTable(int aMaxBucket)
  : HashTableTemplate<int>(aMaxBucket) {}

int IntHashTable::Hash(int& aElem) {
  return aElem % maxBucket;
}

/*********************************************/

ComplexHashTable::ComplexHashTable(int aMaxBucket)
  : HashTableTemplate<Complex>(aMaxBucket) {}

int ComplexHashTable::Hash(Complex& aElem) {
  int h1 = aElem.Re() % maxBucket;
  int h2 = aElem.Im() % maxBucket;
  return (h1 + 19 * h2) % maxBucket;
}
```


Test

```
int main() {
  ComplexHashTable cht(17);
  Complex a[400];
  int k = 0;
  for (int i = 0; i < 2000; i++) {
    Complex elem(rand(), rand());
    if (i % 5 == 0) a[k++] = elem;
    cht.Insert(elem);
  }
  int hits = 0;
  for (int i = 0; i < 400; i++)
    if (cht.Contains(a[i])) hits++;
  cout << "Treffer: " << hits << endl;
}
```

Ausgabe: `Treffer: 400`

Test



Rückblick

- Wir haben spezielle Klassen **`IntList`** und **`HashTable`** (für **`int`**) analysiert und
- verallgemeinert für beliebige Typen mit der Technik der Schablonen.
- Eine Hashtabelle für einen beliebigen Typ erhält man jetzt durch
  - Ableiten von der Schablone,
  - Angabe des Konstruktors (und ggf. des Destruktors),
  - Spezifikation der typ-spezifischen Hash-Funktion.

Achtung:

Eine Schablone muss in einer Übersetzungseinheit **definiert** (oder inkludiert) werden, wenn dort diese Schablone instantiiert wird …

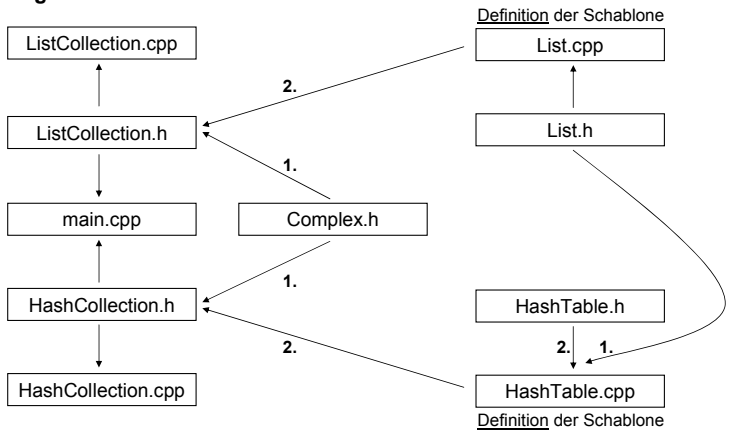
... natürlich **vor** der Instantiierung!

⇒ Organisation des Codes (insbesondere die Inklusionen) gut überdenken!

Beispiel:

`main.cpp`	Hauptprogramm
`Complex.h`	Komplexe Zahlen **`(inline)`**
`ListCollection.*`	Instantiierungen der Schablone **`List`**
`HashCollection.*`	Instantiierungen der Schablone **`HashTable`**
`List.*`	Schablone **`List`**
`HashTable.*`	Schablone **`HashTable`**

Organisation des Codes



Hashing von Binärbäumen

⇒ sinnvoll, da totale Ordnung auf Binärbäumen nicht offensichtlich

```
class BST {
protected:
  BST *fRoot;
private:
  BST *Insert(BST*, int);
public:
  BST() { fRoot = 0; }
  int fValue;
  BST *fLeft, *fRight;
  void Insert(int aValue) {
    fRoot = Insert(fRoot, aValue);
  }
};
```

Annahme:

Die Klasse **BST** (**b**inary **s**earch **t**ree) ist in Bibliothek gegeben.

⇒ Nicht veränderbar!

Aber:

Für Hash-Schablone werden Methoden **`Equal`** und **`Print`** benötigt!

Implementierung (zur Erinnerung)

```
BST *BST::Insert(BST* aTree, int aValue) {
  if (aTree == 0) {
    BST *node = new BST;
    node->fValue = aValue;
    node->fLeft = node->fRight = 0;
    return node;
  }

  if (aTree->fValue > aValue)
    aTree->fLeft = Insert(aTree->fLeft, aValue);
  else if (aTree->fValue < aValue)
    aTree->fRight = Insert(aTree->fRight, aValue);

  return aTree;
}
```

} rekursiv

Wie können Methoden `Equal` und `Print` hinzugefügt werden?

1. Erweiterung der Klasse **BST** um diese Methoden.

   ⇒ Geht nicht, da als Bibliotheksklasse unveränderbar!

2. Ableiten von Klasse **BST** und Methoden hinzufügen.

   ⇒ **Gute Idee!**

Bemerkung:

Hier werden nur Methoden und keine Attribute hinzugefügt.

Die neue Klasse `BinTree`

```
class BinTree : public BST {
private:
  bool Equal(BinTree*, BinTree*);

public:
  BinTree() : BST() { }
  bool Equal(BinTree *aTree) {
    return Equal((BinTree*) fRoot, aTree->GetRoot());
  }
  void Print() { /* to do */ };
  BinTree *GetRoot() { return (BinTree*)fRoot; }
};
```

`cast`-Operation: { **`fRoot`** ist Typ **`BST*`**, wird durch **`cast`** zum Typ **`BinTree*`**

Die neue Klasse `BinTree`

```
bool BinTree::Equal(BinTree* L, BinTree* R) {
  if (L == 0) return (R == 0);
  if (R == 0) return false;
  if (L->fValue != R->fValue) return false;

  BinTree *LL, *LR, *RR, *RL;
  LL = (BinTree*) L->fLeft;
  LR = (BinTree*) L->fRight;
  RL = (BinTree*) R->fLeft;
  RR = (BinTree*) R->fRight;
  return Equal(LL, RL) && Equal(LR, RR);
}
```

Abbruch-bedingungen

cast

Rekursion

Die neue Klasse `BinTree`: Nachtrag

⇒ Eine Methode, die einen Hashwert für einen Baum liefert, wäre nützlich!

public:

```
int Hash() { return Hash((BinTree*)fRoot); }
```

private:

```
int BinTree::Hash(BinTree* B) {
      const c = 275604541; // large prime number
      if (B == 0) return 0;
      int hl = Hash((BinTree*)B->fLeft);
      int hr = Hash((BinTree*)B->fRight);
      int h = (hl * 17 + hr) % c;
      return (h * 23 + B->fValue) % c;
}
```


Instantiierung der Schablone `List` und `HashTable`

```
class BinTreeList : public List<BinTree> {
public:
  BinTreeList();
};
```

ListCollection.h

```
BinTreeList::BinTreeList() : List<BinTree>() {}
```

ListCollection.cpp

```
class BinTreeHashTable : public HashTableTemplate<BinTree> {
public:
      BinTreeHashTable(int aMaxBucket);
      int Hash(BinTree& aElem);
};
```

```
BinTreeHashTable::BinTreeHashTable(int aMaxBucket)
  : HashTableTemplate<BinTree>(aMaxBucket) {}

int BinTreeHashTable::Hash(BinTree& aElem) {
  return aElem.Hash() % maxBucket;
}
```


Rückblick

1. Schablonen für Listen und HashTabellen waren vorhanden
2. Klasse **`BST`** (binary search tree) war in Bibliothek vorhanden
3. Definition der Klasse **`BinTree`** durch Ableiten von **`BST`**
   a) Methode **`Equal`**
   b) Methode **`Print`**
   c) Methode **`Hash`**
4. Instantiieren der Schablonen für Liste und HashTabelle
   a) Definition der Konstruktoren
   b) Methode **`Hash`** (unter Verwendung von **`Hash`** der Klasse **`BinTree`**)

Ergänzungen

- **Anonyme Klassen**

  → Instantiierung einer Schablone ohne Ableiten

  ```
  List<double> dblList; dblList.Insert(23.5);
  ```

  sinnvoll, wenn nur selten oder temporär benutzt;

  lästig, wenn Übergabetyp bei Parametern;

  dann häufige Notlösung: **`typedef`**

  Bsp: **`typedef List<double> DoubleList;`**

- **Funktions-Schablonen**

  →
  ```
  template <class T> void sort(vector<T>&);
  void f(vector<int>&vi, vector<string>& vs) {
    sort(vi); sort(vs);
  }
  ```