Praxisorientierte Einführung in C++
Lektion: "Klassen Templates"

Christof Elbrechter

Neuroinformatics Group, CITEC

June 18, 2014

Table of Contents

Motivation

- Wie auch bei Funktionen existiert eine Vielzahl von Datentypen deren Funktionalität über einen oder mehrere Datentypen abstrahierbar ist
- Damit: Erhöhung der Wiederverwendbarkeit von Code
- **Besonders vorteilhaft:** Templates werden zur Übersetzungszeit aufgelöst $\Rightarrow$ deutlich effizienter also Polymorphismus
- **Aber:** Dennoch ersetzt dieses Polymorphismus nicht
- Beispiel: `Array`-Klasse mit automatischem Speicher-Handling
  - Zunächst für `int`-arrays

int-Array Klasse

```
#include <algorithm>

class IntArray{
   int *data; unsigned int size;
   public:
   IntArray(unsigned int size=0, int initialVal=0):
     data(size ? new int[size] : 0),size(size) {
     std::fill(data,data+size,initialVal);
   }
   IntArray(const IntArray &other):data(0),size(0){ *this = other; }
   IntArray &operator=(const IntArray &other){
     if(data) delete [] data;
     data = other.data ? new int[other.size] : 0;
     size = other.size;
     std::copy(other.data,other.data+size,data);
     return *this;
   }
   ~IntArray(){ if(data) delete []data; }

   int &operator[](unsigned int idx){ return data[idx]; }
   const int &operator[](unsigned int idx) const{ return data[idx]; }
};
```

Template-Array Klasse

```
#include <algorithm>
template<class T>
class Array{
   T *data; unsigned int size;
   public:
   Array(unsigned int size=0, T initialVal=0):
     data(size ? new T[size] : 0),size(size) {
     std::fill(data,data+size,initialVal);
   }
   Array(const Array &other):data(0),size(0){ *this = other; }
   Array &operator=(const Array &other){
     if(data) delete [] data;
     data = other.data ? new T[other.size] : 0;
     size = other.size;
     std::copy(other.data,other.data+size,data);
     return *this;
   }
   ~Array(){ if(data) delete [] data; }

   T &operator[](unsigned int idx){ return data[idx]; }
   const T &operator[](unsigned int idx) const{ return data[idx]; }
};
```

Instanzierungs-Regeln

- Analog zu Funktions-Templates
- Klassen-Templates sind noch keine Klassen ⇒ sie müssen erst Instanziert werden
- implizit oder explizit
- Explizit z.B.: `template class Array<int>;`
- I.d.R. Klassen Templates werden komplett `inline` implementiert
- Dann wird immer alles automatisch implizit instanziert

Instanzierung von STL-Templates

STL ist auch komplett `inline` definiert ⇒ hier muss also niemals irgendwas explizit instanziert werden!

Implementation von Klassen Templates

- Bei `inline`-Implementation kann der Template-Klassenname ohne Typparameter[1] verwendet werden

⇒ Diese sind dann implizit die des aktuellen Templates

```
template<class T>
class Array{
   // ...
   Array &operator=(const Array &other){ ... } // ok!
};
```

- Genau genommen müsste man es so schreiben:

```
template<class T>
class Array{
   // ...
   Array<T> &operator=(const Array<T> &other){ ... } // geht auch!
};
```

[1] es können auch mehrere sein

Separate Deklaration und Implementation

- Bei Trennung von Deklaration und Implementation: **Komplizierter!**

```
template<class T>
class Array{
   // ...
   Array &operator=(const Array &other); // hier nur Deklaration
};

// extern definierte Methoden muessen auch als Template deklariert werden
template<class T>
Array<T> &Array<T>::operator=(const Array<T> &other){
  if(data) delete data;
  data = other.data ? new T[other.size] : 0;
  size = other.size;
  std::copy(other.data,other.data+size,data);
  return *this;
}
```

- Der Typparameter des Funktionsargumentes (`const Array<T> &other`) kann auch weggelassen werden
- Da man sich das allerdings schlecht merken kann, kann man die Typparameter auch immer mit angeben

Klassen-Templates mit Ganzzahlausdrücken

- Wie auch Funktionen können Klassen-Templates auch Ganzzahlausdrücke als Template-Parameter haben
- Nettes Beispiel: Vektor mit fester Länge

Vec.h

```
template<int N,class T>
class Vec{
  T data[N];
public:
  Vec(const T &init=T(0)){
    for(int i=0;i<N;data[i++]=init);
  }
  T &operator[](int idx){ return data[idx];}
  const T &operator[](int idx) const{ return data[idx];}
};
```

- Entspricht Erweiterung eines Fixed-Arrays (komplett auf dem Stack)
- Kann hochgradig effizient übersetzt werden
- Für kleine N: Praktisch keine Schleifen

Template-cast-Operatoren

Vec.h

```
template<int N,class T>
class Vec{
  T data[N];
public:
  // ...
  operator T*(){ return data; }
  operator const T*() const{ return data; }
};
```

- Damit kann eine Vec<T>-Instanz auch immer dort eingesetzt werden wo T-Pointer erwartet werden

```
Vec<100,char> str;
strcpy(str,"Hello World");
std::string s(str);
std::cout << s << std::endl;
```

Rechnen mit Template-Parametern

- Mit ganzzahligen Template-Parametern kann sogar gerechnet werden
- Die Methode cat soll Vektoren unterschiedlicher Dimensionalität konkatenieren
- **Dafür:** Template-Memberfunktion innerhalb eines Templates
- Dieses funktioniert analog zu sonstigen Template-Funktionen

Vec.h

```
template<int N,class T>
  class Vec{ //...
  // template-Methode innerhalb einer Template-Klasse -->
  // der Typparameter muss einen anderen Bezeichner haben
  template<int M>
  Vec<N+M,T> cat(const Vec<M,T> &other) const{
    Vec<N+M,T> v;
    for(int i=0;i<N;i++) v[i]=data[i];
    for(int i=0;i<M;i++) v[i+N]=other[i];
    return v;
  }
};
```

Template-Memberfunktionen

- Bei **Klassen-Templates** können Member-Variablen und -Methoden von den Template-Parametern abhängen
- Das gilt übrigens auch für statische Variablen und Funktionen
- Bei **Template-Memberfunktionen** hingegen hängt nur die eine Funktion von den Template-Parametern ab
- Die Konzepte können sich ergänzen!
- **Anmerkung:** Bei getrennter Deklaration und Implementation von Funktionen wie cat: **Verstand abschalten!**

```
template<int N,class T> template<int M>
Vec<N+M,T> Vec<N,T>::cat(const Vec<M,T> &other) const{
  Vec<N+M,T> v;
  for(int i=0;i<N;i++) v[i]=data[i];
  for(int i=0;i<M;i++) v[i+N]=other[i];
  return v;
}
```

Stream-Operatoren für Klassen-Templates

- Sehr sinnvoll zur Ausgabe/Serialisierung: Implementation des ostream-Operatores (<<)
- Für Klassen-Templates muss auch der ostream-Opearator als Template implementiert werden

```
template<int N,class T>
std::ostream &operator<<(std::ostream &s,const Vec<N,T> &v){
  s << '(';
  for(int i=0;i<N-1;++i){
     s<<v[i]<<',';
  }
  return s<< v[N-1] << ')';
}

int main(){
  Vec<3,float> v3(23);
  Vec<5,float> v5(42);
  std::cout << v3.cat(v5) << std::endl;
}
```

Standard-Typparameter für Klassen-Templates

- Bei Klassen-Templates können Standardwerte auch für Template-Parameter angegeben werden

```
template<int N,class T=float>
class Vec{
   // ...
};
int main(){
   Vec<5> v(4.3); // hier wird implizit 'float' verwendet!
}
```

- **Leider:** Werden Standardwerte für alle Typparameter angegeben, bleibt <> übrig!

```
template<int N=3,class T=float>
class Vec{
   // ...
};
int main(){
   Vec v(0); // Fehler!!
   Vec<> v(0); // ok, aber super-haesslich!
}
```

Ausweg

- Falls es eine *most-common*-Typparameter-Kombination gibt:
  - Allgemeines Template mit allgemeinem Bezeichner definieren
  - `typedef` für den *most-common*-Typ mit speziellem Bezeichner

GenericVec.h

```
template<int N,class T>
class GenericVec{
  // ...
};

typedef GenericVec<3,float> Vec3f;
typedef GenericVec<3,double> Vec3d;
```

- Typen die mittels `typedef` definiert wurden, können dann komplett ohne Angabe von Typparametern verwendet werden

Anmerkung

- Leider können z.Z. mit `typedef` keine eingeschänkten Templates erzeugt werden

```
template<class T>
typedef GenericVec<3,T> Vec3<T>; // waehre schoen, geht aber nicht!
```

- Wird im C++-1x-Standard aber eingeführt werden!

Spezialisierung von Klassen-Templates

- Im Gegensatz zu Template-Funktionen können Klassen-Templates auch teilweise spezialisiert werden
- Hier: Ein praktisches Beispiel: *Wertebereich-Clipping*

Problemfall

```
#include <iostream>

int main(){
  int i = 300;
  unsigned char c = i;
  std::cout << (int)c << std::endl;
}
```

- Ausgabe des des Programms: `44`
- Wertebereich von `unsigned char` ist $0 - 255$
  $\Rightarrow$ `int` wird sozusagen *gesliced*

Clipped-Cast Template

- **Idee:** Definition eines `clipped_cast` Templates, welches den jeweiligen Wertebereich berücksichtigt

clipped_cast.h

```
// Hilfsklasse welche mit Teil-Spezialisierung optimiert werden kann
template<class S,class D>
struct Cast{
  static D clipped(const S &s){
    return static_cast<D>(s); // Standardfall
  }
};

// Diese Funktion soll dann wirklich aufgerufen werden
template<class S,class D>
inline D clipped_cast(const S &s){
   return Cast<S,D>::clipped(s);
}
```

Spezialisierungen der Cast-Klasse

```
// allgemein
template<class S,class D>
struct Cast{
  static inline D clipped(const S &s){
    return static_cast<D>(s);
  }
};

// wenn Quell- und Ziel-Typ identisch: Kein cast notwendig
template<class K>
struct Cast<K,K>{
  static inline K clipped(const K &k){
    return k;
  }
};

// Hier expl. Clipping!
template<>
struct Cast<int,unsigned char>{
  static inline unsigned char clipped(const int &i){
    return (i<0) ? 0 : (i>255) ? 255 : (unsigned char)i;
  }
};
```

Spezialisierungs-Konflikte

- Teilspezialisierugen dürfen nicht gleich-speziell sein
- Hierfür zählt die Anzahl an noch offenen Template-Parametern

```
// da der Wertebereich von double alle anderen beinhaltet:
// kein cast, falls Ziel-Typ double ist (Achtung: Fehler!!)
template<class S>
struct Cast<S,double>{
  static inline double clipped(const S &s){
    return double(s);
  }
};
```

- Damit mehrdeutig:

```
std::cout << clipped_cast<double,double>(5.2) << std::endl;
```

Anmerkungen/Zusammenfassung

- Klassen-Templates sind sehr mächtig, insb. in Kombination mit Template-Funktionen (siehe STL)
- Übung macht den Meister
- STL lässt sich gut verwenden ohne alles im Detail auswendig zu können!
- Bei Schachtelung von Templates muss man manchnmal aufpassen:

```
Array<Vec<3,float>> a; // FEHLER??
```

- Problem: Übersetzer denkt >> ist ein istream-Operator
- Lösung: Extra-Whitespace

```
Array<Vec<3,float> > a; // nun geht's!
```