Praxisorientierte Einführung in C++
Lektion: "Funktions-Templates"

Christof Elbrechter

Neuroinformatics Group, CITEC

May 15, 2014

Table of Contents

Funktionen ...

- Bisher: Abstraktion über Werte

Beispiel: Abstraktion über die Werte der Variablen a und b

```
void add(int a, int b){
   return a+b;
}
```

- *Variable/Container*-Modell: Abstraktion über den Inhalt eines Containers
- Nun: **Abstraktion über *Typen***
- *Variable/Container*-Modell: Abstraktion über den Typ des Containers

Funktionen ...

- Bisher: Abstraktion über Werte

Beispiel: Abstraktion über die Werte der Variablen a und b

```
void add(int a, int b){
   return a+b;
}
```

- *Variable/Container*-Modell: Abstraktion über den Inhalt eines Containers
- Nun: **Abstraktion über *Typen***
- *Variable/Container*-Modell: Abstraktion über den Typ des Containers
- **Templates sind cool!**

Typisches Problem

- Einige Funktionen sind eigentlich (größtenteils) unabhängig vom Typ
- Beispiele: `swap`, `sort`, `for_each`, usw.
- Funktionsüberladung: möglich, Funktion muss aber für jeden Typ implementiert werden

Typisches Problem

- Einige Funktionen sind eigentlich (größtenteils) unabhängig vom Typ
- Beispiele: `swap`, `sort`, `for_each`, usw.
- Funktionsüberladung: möglich, Funktion muss aber für jeden Typ implementiert werden
  - Aufwendig
  - Unübersichtlich
  - Hoher Wartungsaufwand
  - Kann nicht im Ramen einer Bibliothek angeboten werden, da nicht alle potentiellen Typen bekannt sind

Swap als überladene Funktion

swap_overloading.h

```
// zunaechst fuer ints
inline void swap(int &a, int &b){
   int tmp = a;
   a = b;
   b = tmp;
}
// .. und fuer floats (muehsam)
inline void swap(float &a, float &b){
   float tmp = a;  a = b;  b = tmp;
}
```

Swap als überladene Funktion

swap_overloading.h

```
// zunaechst fuer ints
inline void swap(int &a, int &b){
   int tmp = a;
   a = b;
   b = tmp;
}
// .. und fuer floats (muehsam)
inline void swap(float &a, float &b){
   float tmp = a;  a = b;  b = tmp;
}
```

```
// ein Macro koennte helfen
#define SWP_BODY(T) T tmp=a;a=b;b=tmp
inline void swap(char &a, char &b) { SWP_BODY(char); }
inline void swap(char* &a, char* &b) { SWP_BODY(char*); }
// .. oder gleich die ganze Funktion als Macro
#define SWP_FUNC(T) inline void swap(T &a, T &b) { SWP_BODY(T); }
SWP_FUNC(unsigned char)
SWP_FUNC(double)
```

Swap als überladene Funktion

- Es existiert nicht *die* Lösung
- Auch z.B. möglich: SWAP-Makro ganz ohne Funktion

```
#define SWAP(T,A,B) { T tmp=A; A=B; B=tmp; }
```

- Typ muss immer explizit mit angegeben werden
- Kann aufgrund der Klammern auch nicht überall verwendet werden[1]
- Bei komplizierteren Funktionen: Code-Bloat
- Compiler kann nicht über inlining entscheiden

[1] ohne Klammer dürfte man die Funktion allerdings nur einmal pro Block verwenden

Swap als überladene Funktion

- Es existiert nicht *die* Lösung
- Auch z.B. möglich: SWAP-Makro ganz ohne Funktion

```
#define SWAP(T,A,B) { T tmp=A; A=B; B=tmp; }
```

- Typ muss immer explizit mit angegeben werden
- Kann aufgrund der Klammern auch nicht überall verwendet werden[1]
- Bei komplizierteren Funktionen: Code-Bloat
- Compiler kann nicht über inlining entscheiden
- **Ausweg: Templates**

[1] ohne Klammer dürfte man die Funktion allerdings nur einmal pro Block verwenden

Swap als Template

swap.h

```
template<class T>
inline void swap(T &a, T &b){
   T tmp = a;
   a = b;
   b = tmp;
}
```

- Das swap-Template ist gültig für alle Typen

Swap als Template

swap.h

```
template<class T>
inline void swap(T &a, T &b){
   T tmp = a;
   a = b;
   b = tmp;
}
```

- Das swap-Template ist gültig für alle Typen

Einzige Bedingung

- Die Typen müssen die verwendeten Funktionalitäten bereitstellen
  - T muss einen *kopierbar* und *zuweisbar* sein

Template-Instanzierung

- **Aber:** Template-Definition erzeugt noch kein gültiges Symbol
- Ein Template ist lediglich eine *Schablone* für eine Funktion

Template-Instanzierung

- **Aber:** Template-Definition erzeugt noch kein gültiges Symbol
- Ein Template ist lediglich eine *Schablone* für eine Funktion
- Instanzierung kann *explizit* oder *implizit* erfolgen

Template-Instanzierung

- **Aber:** Template-Definition erzeugt noch kein gültiges Symbol
- Ein Template ist lediglich eine *Schablone* für eine Funktion
- Instanzierung kann *explizit* oder *implizit* erfolgen

Instanzierungstechniken

- implizit: D.h. Fuktionsdefinition muss `inline` sein, ansonsten: einfach
- explizit: Verhältnismäßig kompliziert

Template-Instanzierung

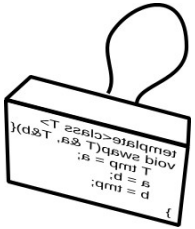
- Funktions-Templates = Schablonen
- I.d.R werden Templates `inline` definiert
- `inline`-deklarierte Templates müssen nicht explizit instanziert werden, da hier ja kein Symbol entsteht

Template-Instanzierung

- Funktions-Templates = Schablonen
- I.d.R werden Templates `inline` definiert
- `inline`-deklarierte Templates müssen nicht explizit instanziert werden, da hier ja kein Symbol entsteht
- Instanzierung von nicht-`inline`-Templates:
  - implizit durch Verwendung
  - explizit mittels eines speziellen Instanzierungsausdrucks

Template-Instanzierung

- Funktions-Templates = Schablonen
- I.d.R werden Templates `inline` definiert
- `inline`-deklarierte Templates müssen nicht explizit instanziert werden, da hier ja kein Symbol entsteht
- Instanzierung von nicht-`inline`-Templates:
  - implizit durch Verwendung
  - explizit mittels eines speziellen Instanzierungsausdrucks
  - exportieren mittels `export`-Schlüsselwort (geht nicht!)



- explizit bedeutet hier: Ausdrückliches Abstempeln des Template-Stempels

Beispiel für Templateinstanzierung

- Es soll ein Funktionstemplate `arrayCopy` implementiert werden
- `arrayCopy-Funktion` kopiert Daten aus einem zusammenhängenden Speicherbereich in einen Anderen

Beispiel für Templateinstanzierung

- Es soll ein Funktionstemplate `arrayCopy` implementiert werden
- `arrayCopy-Funktion` kopiert Daten aus einem zusammenhängenden Speicherbereich in einen Anderen

Annahme

- `arrayCopy` soll zunächst *nur für einige Typen* definiert werden
- Da das eigentliche Kopieren eines Arrays i.d.R. einige Prozessorzyklen benötigt, bringt inline-Implementation keinen grundsätzlichen Geschwindigkeitsvorteil

Beispiel arrayCopy

array_copy.h

```
template<typename T>
void arrayCopy(const T *src, T *dst, int len);
```

Beispiel arrayCopy

array_copy.h

```
template<typename T>
void arrayCopy(const T *src, T *dst, int len);
```

array_copy.cpp

```
template<typename T>
void arrayCopy(const T *src, T *dst, int len){
   for(int i=0;i<len;++i){
      dst[i] = src[i];
   }
}
```

Beispiel arrayCopy

array_copy.h

```
template<typename T>
void arrayCopy(const T *src, T *dst, int len);
```

array_copy.cpp

```
template<typename T>
void arrayCopy(const T *src, T *dst, int len){
   for(int i=0;i<len;++i){
      dst[i] = src[i];
   }
}
```

```
//Explizite Instanzierung fuer int, float und double
template void arrayCopy(const int*,int*,int);
template void arrayCopy(const float*,float*,int);
template void arrayCopy(const double*,double*,int);
```

Verwendung von Funktions-Templates

- Template Funktionen werden im einfachsten Fall *normal* verwendet

Beispiel

```
#include <array_copy.h>

int main(int n, char **argv){
   int a[10]={0,1,2,3,4,5,6,7,8,9};
   int *b = new int[20];
   arrayCopy(a,b,10);
   arrayCopy(a,b+10,10);
}
```

- Template-Typ wird – falls möglich – automatisch inferiert
  ($\Rightarrow$ dazu später mehr)

Template-Instanzierung

- Für nicht explizit-instanzierte Versionen von `arrayCopy` fehlen die Symbole!

Beispiel

```
#include <array_copy.h>

int main(int n, char **argv){
   char **argvCpy = new char*[n];
   arrayCopy(argv,argvCpy,n);
}
```

Template-Instanzierung

- Für nicht explizit-instanzierte Versionen von `arrayCopy` fehlen die Symbole!

Beispiel

```
#include <array_copy.h>

int main(int n, char **argv){
   char **argvCpy = new char*[n];
   arrayCopy(argv,argvCpy,n);
}
```

- Lässt sich korrekt übersetzen
- Beim Linken fehlt allerdings die `arrayCopy`-Version mit `T=char*`

Template-Instanzierung

- Für nicht explizit-instanzierte Versionen von `arrayCopy` fehlen die Symbole!

Beispiel

```
#include <array_copy.h>

int main(int n, char **argv){
   char **argvCpy = new char*[n];
   arrayCopy(argv,argvCpy,n);
}
```

- Lässt sich korrekt übersetzen
- Beim Linken fehlt allerdings die `arrayCopy`-Version mit `T=char*`
- **Aber:** Fehlende `arrayCopy` Version kann hier nicht *nach-instanziert* werden
- Implementation ist nur in `array_copy.cpp`-File verfügbar

Template-Instanzierung

Versuch

```
#include <arrayCopy.h>
template void arrayCopy(char**,char**,int); // ( geht nicht! )
```

- explizite Instanzierung muss in `array_copy.cpp` nachträglich ergänzt werden

Template-Instanzierung

Versuch

```
#include <arrayCopy.h>
template void arrayCopy(char**,char**,int); // ( geht nicht! )
```

- explizite Instanzierung muss in `array_copy.cpp` nachträglich ergänzt werden
- `inline` im Header deklarierte Templates können hingegen direkt verwendet werden

array_copy.h

```
template<typename T>
inline void arrayCopy(T *src, T *dst, int len){
   for(int i=0;i<len;++i){
      dst[i] = src[i];
   }
}
```

Implizite vs explizite Instanzierung

- Instanzierung erfolgt in diesem Fall also implizit
- Da die Funktion `inline` ist, entscheidet der Compiler ob die Funktion tatsächlich `inline` eingebaut wird (z.B im Falle von swap sinnvoll), oder jede Instanz des Templates irgendwie zu einem *anonymen Symbol verpackt* wird
- Dieses könnte mehrfach verwendet werden, falls das Typ-gleiche Template mehrfach innerhalb einer Translation-Unit verwendet wird
- I.d.R. werden Templates im Header definiert, so dass eine implizite Instanzierung möglich ist
- Hilfsfunktionen können natürlich auch Templates sein
  - Die deklariert man dann am besten `static` oder ebenfalls `inline`
  - Implizite Instanzierung ist dann nur in dem Source-File möglich
  - Die Symbole werden allerdings (wegen `static` oder wegen `inline`) nicht nach außen gelinkt (sinnvoll für Hilfsfunktionen)

Selektion eines bestimmten Templates

- Templates müssen sich nicht immer in der Parameter-Liste unterscheiden

parse.h

```
#include <sstream>
template<class T>
T parse(const std::string &s){
  std::istringstream str(s);
  T t; s >> t;  return t;
}
```

Selektion eines bestimmten Templates

- Templates müssen sich nicht immer in der Parameter-Liste unterscheiden

parse.h

```
#include <sstream>
template<class T>
T parse(const std::string &s){
  std::istringstream str(s);
  T t; s >> t;  return t;
}
```

- Doch, welches Template ist gemeint?

main.cpp

```
int main(int n, char **argv){
   int a = parse(argv[1]); // aber fuer welches T?
   float a = parse(argv[2]); // hier auch?
}
```

Selektion eine bestimmten Templates

- Wie bei der Funktionsüberladung kann ein lvalue nicht bei der Typ-Inferenz helfen[2]
- Ausweg: Explizites Auswählen einer speziellen Template-Ausprägung

main.h

```
int main(int n, char **argv){
   int a = parse<int>(argv[1]);
   float b = parse<float>(argv[2]);

   // oder mit anschl. Typumwandlung
   float a = parse<int>(argv[2]);
}
```

[2] Es gibt allerdings eine Möglichkeit dieses zu erreichen (⇒ später)

Instanzierung selektierter Templates

- Dieser Mechanismus muss auch für explizite Instanzierungen verwendet werden, wenn der Typ nicht aus der Parameter-Liste der Funktion inferiert werden kann

parse.h

```
template<typename T>
T parse(const std::string &t);
```

Instanzierung selektierter Templates

- Dieser Mechanismus muss auch für explizite Instanzierungen verwendet werden, wenn der Typ nicht aus der Parameter-Liste der Funktion inferiert werden kann

parse.h

```
template<typename T>
T parse(const std::string &t);
```

parse.cpp

```
#include <parse.h>

template<typename T>
T parse(const std::string &t) {
   std::istringstream str(s);
   T t; s >> t;  return t;
}
// explizite Instanzierung mit Angabe des Typs
template int parse<int>(const std::string);
template float parse<float>(const std::string);
```

Spezialtechnik für lvalue-basierte Typ-Inferenz

Vorausblick:

- Wie im vorherigen Beispiel für `parse` erläutert: lvalue kann nicht für die Typ-Inferenz verwendet werden
- Ein kleiner Trick ermöglicht dieses aber dennoch:
- Wenn der C++-Compiler auf einen Ausdruck `A=B` trifft, sucht er automatisch nach Zuweisungsmöglichkeiten
  - Ist `Typ(A) == Typ(B)` und `Typ(A)` ist POD-Typ $\Rightarrow$ trivial
  - Gibt es einen Zuweisungsoperator `A::operator=(const B&)`
  - Kann `B` implizit zu etwas umgewandelt werden, was `A` zugewiesen werden kann (z.B. `int i = true;`)
- Um Zuweisung zu ermöglichen
  - Zuweisungsoperator des lvalue Typen überladen (geht nur für eigene Klassen)
  - Impliziten cast-operator des rvalue Typen überladen (geht auch nur für eigene Klassen; dies ist hier aber meisst ausreichend)

Spezialtechnik für lvalue-basierte Typ-Inferenz

parse_deluxe.h

```
#include <iostream>
#include <sstream>
struct ParsedString{ // Ok, wir haben noch gar keine Klassen
                     // dann halt ein kleiner Vorausblick !
  std::string s;
  ParsedString(const std::string &s):s(s){}
  template<class T> inline operator T() const {
    std::istringstream str(s);
    T t; str >> t; return t;
  }
};
ParsedString parse(const std::string &s) {
  return ParsedString(s);
}
int main(int n, char **ppc){
  int i = parse(ppc[1]);
  float f = parse(ppc[2]);
  std::string s = parse(ppc[3]);
}
```

Mehrere Typ-Parameter

- Templates können auch über mehrere Typen parametrisiert sein

less_than.h

```
template<typename T1, typename T2>
inline bool lt(const T1 &t1,const T2 &t2){
   return a < b;
}
```

- In Einzelfällen können die Typ-Parameter auch nach außen gar nicht sichtbar sein

is_a.h

```
template<typename T>
inline bool is_a(const std::string &text){
   try{
      parse<T>(text); // wirft Exception bei Fehler
      return true;
   }catch(ParsingException &ex){  return false; }
}
```

Templates von Templates

```
#include <vector>
#include <list>

template<class A,class B,class D,
         template<typename,typename> class C>
void push(C<B,D> &c, const A &a){
  c.push_back(static_cast<B>(a));
}
int main(){
  std::vector<int> v;
  push(v,2.3);
  push(v,false);

  std::list<float> l;
  push(l,false);
  push(l,0);
  push(l,0.3f);
}
```

- Gibt's auch!

Erweiterte Techniken

- Template Spezialisierung
  - Allgemeines Template kann für bestimmte Typen spezialisiert werden
  - D.h. es kann eine völlig andere Implementation angegeben werden
  - Sinnvoll, falls allgemeines Template für alle bis auf einige Ausnahmen gültig ist
    - Beispiel: String-Konvertierungsfunktion `str(T)` ($\Rightarrow$ kommt gleich)
    - Rekursionsbasis bei rekursiven Templates ($\Rightarrow$ das auch!)

Erweiterte Techniken

- Template Spezialisierung
  - Allgemeines Template kann für bestimmte Typen spezialisiert werden
  - D.h. es kann eine völlig andere Implementation angegeben werden
  - Sinnvoll, falls allgemeines Template für alle bis auf einige Ausnahmen gültig ist
    - Beispiel: String-Konvertierungsfunktion `str(T)` ($\Rightarrow$ kommt gleich)
    - Rekursionsbasis bei rekursiven Templates ($\Rightarrow$ das auch!)
- Template-Abstraktion über
  - Ganzzahltypen
  - Rekursive Templates
  - Funktionstypen

Template Spezialisierung (Beispiel String-Konvertierung)

- Konvertierungsfunktion `str` soll beliebigen Datentypen in einen `std::string` konvertieren

str.h

```
#include <iostream>
#include <sstream>
#include <string>

// Basisimplementation, zunaechst verwendet fuer alle Typen T
template<class T>
inline std::string str(const T &t){
  std::ostringstream str;
  str << t;
  return str.str();
}
```

- `str` kann einem nun schon das Leben erheblich erleichtern

Template Spezialisierung (Beispiel String-Konvertierung)

Anwendungsbeispiel:

```
#include <fstream>
#include <str.h>
int main(){
  /// read 100 numbered files ...
  for(int i=0;i<100;++i){
    std::ifstream s( ("file-"+str(i)+".txt").c_str() );
    if(s){ ... }
  }
}
```

Template Spezialisierung (Beispiel String-Konvertierung)

Anwendungsbeispiel:

```
#include <fstream>
#include <str.h>
int main(){
  /// read 100 numbered files ...
  for(int i=0;i<100;++i){
    std::ifstream s( ("file-"+str(i)+".txt").c_str() );
    if(s){ ... }
  }
}
```

- Doch was passiert bei anderen Typen?
- Evtl. werden manche Typen nicht wie gewollt von dem `std::ostringstream` in `str` behandelt
  - `unsigned char` wird nicht als Zahl-Typ, sondern als character interpretiert
  - `std::ostringstream`-basierte Implementation ist sehr ineffizient für `std::string` und `char*`

Template Spezialisierung (Beispiel String-Konvertierung)

age_output.cpp

```
typedef unsigned char Age;

int main(){
   std::string name = "Florian P. Schmidt";
   Age age = 77;
   std::string text = name + " ist " + str(age) + " Jahre alt!";
   std::cout << text << std::endl;
}
```

```
> ./age_output
Florian P. Schmidt ist M Jahre alt!
```

Template Spezialisierung (Beispiel String-Konvertierung)

- Da es sich bei `unsigned char` um einen Character-Typen handelt, wird sein Wert als `char` ausgegeben
- Wir wollten `unsigned char` allerdings als numerischen Typen verwenden
  - 1.Ausweg: Anderen Typ verwenden (Hier noch vertretbar, i.A. aber suboptimal)
  - 2.Ausweg: `str`-Template für `unsigned char` spezialisieren

Template Spezialisierung

```
template<>
inline std::string str(const unsigned char &x){
   std::ostringstream str;
   str << (int)x;
   return str.str();
```

Template Spezialisierung (Beispiel String-Konvertierung)

- Tatsächlich kann auch einfach auf einen anderen Typ verwiesen werden

Template Spezialisierung

```
template<>
inline std::string str(const unsigned char &x) {
  return str<int>(x);
}
```

- Ausgabe ist nun (beinahe) korrekt!

```
> ./age_output
Florian P. Schmidt ist 77 Jahre alt!
```

Template Spezialisierung (Beispiel String-Konvertierung)

- Template-Spezialisierungen können auch zur Optimierung verwendet werden
- Implementation mittels `std::ostringstream` ist für manche Typen etwas kompliziert:

```
// strings koennen einfach kopiert werden
template<>
inline std::string str(const std::string &x){
   return x;
}
// char* auch
template<>
inline std::string str(const char *const &p){
   return std::string(p);
}
```

Weitere Informationen

- Die Spezialisierung muss nach der Deklaration des eigentlichen Templates erfolgen
  - 1.) Allgemeine Template Definition
  - 2.) Spezialisierung(en)
  - 3.) Explizite Instanzierungen oder implizite Verwendung bei `inline`-definierten Templates

Weitere Informationen

- Die Spezialisierung muss nach der Deklaration des eigentlichen Templates erfolgen
  - 1.) Allgemeine Template Definition
  - 2.) Spezialisierung(en)
  - 3.) Explizite Instanzierungen oder implizite Verwendung bei `inline`-definierten Templates
- Übersetzer wählt immer die speziellste Implementation aus

Weitere Informationen

- Die Spezialisierung muss nach der Deklaration des eigentlichen Templates erfolgen
  - 1.) Allgemeine Template Definition
  - 2.) Spezialisierung(en)
  - 3.) Explizite Instanzierungen oder implizite Verwendung bei `inline`-definierten Templates
- Übersetzer wählt immer die speziellste Implementation aus
- Funktions-Templates werden immer vollständig spezialisiert
- D.h. im Falle mehrerer Typ-Parameter müssen bei Spezialisierungen immer alle Typ-Parameter festgelegt werden
  - in C++-11 sollte es ursprünglich auch mal "partial specializations" für Funktionen geben
  - gibts aber nicht!

Teil-Spezialisierung von Templates

- Teil-Spezialisierugen: Z.Z. nicht möglich

nicht möglich

```
template<class S, class D>
D convert(const S &s){
  return (D)s;
}
// ungueltiger Spezialisierungsversuch
template<class S>
std::string convert<S,std::string>(const S &s){
   return str(s);
}
```

- Später:
  - Klassen Templates
  - Da geht sowas dann :-)

Abstraktion über Ganzzahltypen

- Templates können auch über Ganzzahltypen abstrahieren
- 1.Beispiel: Die Funktion `power` soll die N-te Potenz einer Zahl ausgeben
- **Problem:** Allgemeine `power`-Funktion ist relative Aufwendig da:
  - ... sie mit beliebigen Exponenten klar kommen muss
  - ... Exponenten auch nicht-Ganzzahlen sein können
- Insbesondere bei niedrigen ganzzahligen Exponenten (0,1,2,usw.) kann mittels '*' eine viel effizientere Implementation erbracht werden

Templates mit Ganzzahltypen

- Hier eine für die Exponenten 0-4 optimierte power-Implementation

Template pow mit int als Typparameter

```
#include <iostream>
#include <cmath>

template<int N>
inline double power(double x){
  switch(N){
    case 0: return 1;
    case 1: return x;
    case 2: return x*x;
    case 3: return x*x*x;
    case 4: return x*x*x*x;
    default: return ::pow(x,N);
  }
}
```

Anmerkungen zu dieser Implementation von power

- Großer Vorteil: Da N konstant ist, kann der Compiler direkt den richtigen Ausdruck auswählen
- `switch`-Statement kann weg-optimiert werden
- Nachteile:
  - Viel Schreibarbeit
  - Optimiert nur bis zu bestimmtem Exponenten
  - Geht nur, falls der Exponent konstant ist

Aufrufen von power

```
#define EXP1 7
const int EXP2 = 4;
// fuer Konstante Ausdruecke: ok
double x = power<EXP1>(4.3) + power<EXP2>(4.2) + power<2>(2);
for(int i=0;i<5;++i) {
   x += power<i>(2); // Fehler (i ist kein konstanter Ausdruck)
```

Benchmark

- Wie groß ist der Geschwindigkeitsvorteil?
- Die neue power Implementation ist deutlich schneller als pow aus dem <cmath>-Header
- kompiliert man mit dem g++ mit dem Optimierungs-Flag -O3, so ist power ca. 5-mal schneller

Rekursive Templates

- (Hier) auch mit Ganzzahl-Typen
- Beispiel: fibonacci-Funktion (Schritt für Schritt)

Erinnerung: Fibonacci-Reihe

```
fib(n)=fib(n-1)+fib(n-2)
fib(0)=0
fib(1)=1
```

Rekursive Templates

- (Hier) auch mit Ganzzahl-Typen
- Beispiel: fibonacci-Funktion (Schritt für Schritt)

Erinnerung: Fibonacci-Reihe

```
fib(n)=fib(n-1)+fib(n-2)
fib(0)=0
fib(1)=1
```

Naive rekursiver Algorithmus

```
unsigned int fib(unsigned int N){
  return N < 2 ? N : fib(N-1)+fib(N-2);
}
```

- **Problem:** Wachsender Rekursions-Stack macht Funktion sehr langsam

Fibbonacci Zahlen mittels rekursiven Template

- Zunächst ein naiver Ansatz

Naiver Ansatz

```
// Template definition: ok
template<unsigned int N>
unsigned int fib(){
  return N > 2 ? fib<N-1>()+fib<N-2>() : N;
}
int main(){
  // Aufruf des Templates fuehrt aber zu unendlicher Rekursion
  std::cout << fib<5>() << std::endl;
}
```

Fibbonacci Zahlen mittels rekursiven Template

- Rekursions-Basis **muss** mittels Spezialisierung definiert werden

Fibbonacci Zahlen mittels rekursiven Template

- Rekursions-Basis **muss** mittels Spezialisierung definiert werden

Fibonacci-Zahlen mittels rekursivem Template

```
template<unsigned int N>
unsigned int fib(){
  return N > 2 ? fib<N-1>()+fib<N-2>() : N;
}
// Festlegen der Rekursionsbasen
template<> unsigned int fib<0>(){ return 0; }
template<> unsigned int fib<1>(){ return 1; }
```

Fibbonacci Zahlen mittels rekursiven Template

- Rekursions-Basis **muss** mittels Spezialisierung definiert werden

Fibonacci-Zahlen mittels rekursivem Template

```
template<unsigned int N>
unsigned int fib(){
  return N > 2 ? fib<N-1>()+fib<N-2>() : N;
}
// Festlegen der Rekursionsbasen
template<> unsigned int fib<0>(){ return 0; }
template<> unsigned int fib<1>(){ return 1; }
```

Funktionsaufruf

```
unsigned int x = fib<22>();
```

Fibbonacci Zahlen mittels rekursiven Template

- Rekursions-Basis **muss** mittels Spezialisierung definiert werden

Fibonacci-Zahlen mittels rekursivem Template

```
template<unsigned int N>
unsigned int fib(){
  return N > 2 ? fib<N-1>()+fib<N-2>() : N;
}
// Festlegen der Rekursionsbasen
template<> unsigned int fib<0>(){ return 0; }
template<> unsigned int fib<1>(){ return 1; }
```

Funktionsaufruf

```
unsigned int x = fib<22>();
```

- Argument muss allerdings immer ein konstanter Ganzzahlausdruck sein
- Dafür kann Übersetzer den Template-Ausdruck zur Compile-Zeit *völlig* ausrollen

Benchmark

- Mit entsprechenden Optimierungen (g++ mittels `-O3 -march=native`) kann der Aufruf `fib<22>()` weiter optimiert werden
- Denkbar wäre z.B. eine Auswertung zu einem Ganzzahl-Literal
- Wird aber beim g++ nicht gemacht :-(
- Dennoch erheblich performanter
- `fib`-Template ist u.U. über 10^6 mal schneller !!! (ermittelt für `fib<25>()`)

Fibbonacci-Zahlen Iterativer Algorithmus

- Vollständigkeitshalber: `fib` kann natürlich auch iterativ implementiert werden:

Fibbonacci-Zahlen iterativ

```
unsigned int fib_iterative(unsigned int N){
  unsigned int f[2] = {0,1};
  while(N--){
    unsigned int next = f[0]+f[1];
    f[0] = f[1];
    f[1] = next;
  }
  return f[0];
}
```

- `fib_iterative` ist nochmal fast doppelt so schnell wie die Template-Variante

Apropos Rekursion ...

- Natürlich kann man ja auch power rekursiv definieren
- Wie zuvor: Rekursionsbasis wird mittels Template-Spezialisierung definiert

Rekursives **power**-Template

```
template<int N>
inline double power(double x){
  return x*power<N-1>(x);
}

template<>
inline double power<0>(double x){
  return 1;
}
```

- Genau so schnell wie das erste power-Template
- Ab einem Exponenten von ca. 50 werden die power-Templates langsamer als die standard-Funktion pow

Templates über Funktionstypen

- Template-Typ-Parameter können auch Funktionstypen sein
- Damit lässt sich bspw. `for_each` noch effizienter implementieren

Erinnerung naives for_each

```
void for_each(int *array, int len, void (*f)(int &)){
   for(int i=0;i<len;++i){
      f(array[i]);
   }
}
```

Templates über Funktionstypen

- Template-Typ-Parameter können auch Funktionstypen sein
- Damit lässt sich bspw. `for_each` noch effizienter implementieren

Erinnerung naives for_each

```
void for_each(int *array, int len, void (*f)(int &)){
   for(int i=0;i<len;++i){
      f(array[i]);
   }
}
```

- **Großer Nachteil:** Funktion kann nur schwer *geinlined* werden, da Funktionszeiger kein konstanter Ausdruck ist

for_each mittels Template

for_each-Template

```
template<class T, void (*f)(T&)>
void for_each(T *array, int len){
   for(int i=0;i<len;++i){
      f(array[i]);
   }
}
```

for_each mittels Template

for_each-Template

```
template<class T, void (*f)(T&)>
void for_each(T *array, int len){
   for(int i=0;i<len;++i){
      f(array[i]);
   }
}
```

- Nun: f ist zur Übersetzungszeit konstant und kann `inline` übersetzt werden (Großer Vorteil für sehr kurze Funktionen `f`)
- Template-Version von `for_each` kann wesentlich besser optimiert werden
- Außerdem: Beliebiger Array-Typ
- Immer noch fest allerdings: `void(*f)(T&)`

for_each noch besser

for_each

```
template<class T,class Func>
void for_each(T *array, int len, Func f){
   for(int i=0;i<len;++i){
      f(array[i]);
   }
}
```

- Funktions-Typ wird automatisch inferiert
- Typ `Func` *matched* auch auf Funktionstypen
- Aufruf ohne spitze Klammern möglich
- Geht für alle `Func f` bei denen daraus gültiger Code entsteht
  ($\Rightarrow$ **Duck-Typing**)

for_each noch besser

for_each

```
template<class T,class Func>
void for_each(T *array, int len, Func f){
   for(int i=0;i<len;++i){
      f(array[i]);
   }
}
```

- Funktions-Typ wird automatisch inferiert
- Typ `Func` *matched* auch auf Funktionstypen
- Aufruf ohne spitze Klammern möglich
- Geht für alle `Func f` bei denen daraus gültiger Code entsteht
  ($\Rightarrow$ **Duck-Typing**)
- Wie wir später sehen werden, können hier auch sog. Funktoren übergeben werden