C++ Templates

- Wozu Templates?
- Definition von Templates
- Gebrauch von Templates
- Instanziierung und Organisation

Birgit Möller & Denis Williams
AG Bioinformatik & Mustererkennung
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Wozu Templates?

- Generizität
  - Klassen und Funktionen, die von *Typen* abhängen
  - Funktionen denen Funktionen Übergeben werden (*ohne* Laufzeitkosten) (z.B. Vergleiche)
    Das nennt sich *Polymorphismus* (*Übersetzungs-, Compile-Time-* oder *Parameter-Polymorphismus* im Gegensatz zu *Laufzeit-Polymorphismus*, der durch virtuelle Funktionen entsteht)

- Geschwindigkeit
  - Wird komplett während der Compilierung aufgelöst.
  - So als ob man die Klassen von Hand geschrieben hätte.

Wozu Templates?

```
template <class matrixelement>
class matrix
{
 public:
  matrixelement getElem(int x, int y) { return m_rows[y][x]; }
  void setElem(int x, int y, matrixelement value) { m_rows[y][x] = value; }

 private:
  matrixelement **m_rows;    ///< Pointer to each row of the matrix.
};


matrix<int> mint;
matrix<double> mdouble;

class complex;
matrix<complex> mcomplex;

int value = 200;
mint.setElem(10,20, value);
```

- Typäquivalenz
  Benutzung wie normale Typen

Wozu Templates?

- Typprüfung und Instanziierung
  Bei der Template-Instanziierung werden (mindestens) alle benötigten Methoden der Template Klasse instanziiert (es entstehen sog. *Spezialisierungen* des Templates für ein spezielles Argument.
  Dann werden auch Typprüfungen durchgeführt (z.B. auch Anforderungen wie T->methode())

```
Definition:
template<class matrixelement>
void matrix<matrixelement>::printElem(int x, int y)
{
  getElem(x,y).print();
}

mint.print(10,20);
```

- Typprüfung ergibt, dass der Typ *matrixelement* eine methode *print()* haben muss!

Definition von Templates

- Das *template*-Schlüsselwort

```
template<class elem, int layer = 1>
class image
{
...
}
```

muß mit *template*< beginnen und endet mit dem nächsten ungeschachtelten >

- Parameter
  - Bsp: image von int,float... mit verschiedenen anzahlen von Ebenen
  - Parameter können sein:
    - *class* (oder *typename*) - für allgemeine Typen
    - einfache Typen - wie int,float
    - Adressen (Pointer?!) - von Objekten oder Funktionen
    - Funktionsnamen (Übergabe von Funktionen OHNE Laufzeitkosten!)
  - Default Parameter sind auch möglich

Gebrauch von Templates

- impliziter Gebrauch
  wenn klar ist, welches Template benutzt wird, braucht man das temlate nicht kenntlich machen

```
template<class T> int g(matrix<T> &m) { ... }

matrix<int> m;
g(m);
```

Typ ist *int g<int>(matrix<int> &);*

- expliziter Gebrauch

```
  tempate<class T> T erzeugeCopie(T) { ... }

  int zahl = erzeuge(20.0);
  erzeuge<int> oder erzeuge<float>?

deshalb:
  int zahl = erzeuge<int>(20.0);

  template<class T> class meineKlasse {...}
  meineKlasse<double> k;
```

Gebrauch von Templates

- Überladen
  Wie andere Funktionen auch.
  Es können leicht Mehrdeutigkeiten auftreten -> durch expliziten Gebrauch lösen.

- Typedefs und Typenames
  Typedefs sind Typ-Synonyme, damit leichter lesbar

```
template<class imageElement, int layers>
class image
{
...
};

typedef image<unsigned char,1> grayImage;
typedef image<unsigned char,3> colorImage;
```

Gebrauch von Templates

- auch für gleichlautenden Code benutzbar:

```
 template<class T> class myTemp1 { typedef T* iterator; ... };
                   class myTemp2 { typedef int* iterator; ... };

Damit kann man z.B. folgendes schreiben:

 template<class C> void f(C & c)
 {
    C::iterator i; ...
 }
```

dabei kann der compiler durcheinander kommen und nicht wissen, daß *iterator* ein Typ sein soll (er kennt ja *C* nicht vorher) deshalb muss man dabei *C::iterator* explizit als Typ deklarieren:

```
typename C::iterator i;
```

Instanziierung und Organisation

- Template Instanziierung beim g++
  Wie stellt der Compiler sicher, daß die entsprechende Spezialisierug an der richtigen Stelle zur Verfügung steht, und nicht doppelt vorkommt?
  - Borland Model
    Compiler erzeugt Template Instanziierungen, wo immer sie benötigt werden. Der Linker erkennt dabei doppelte Instanziierungen und fasst sie zusammen.

    Vorteile: linker braucht nur .o files zu betrachten
    Nachteil: hohe Compile-Time weil template code mehrfach übersetzt wird
    Folge: man muss beim Erzeugen von object-files die template DEFINITION zur Verfügung stellen -> üblicherweise wird die Template-Definition in den Header geschrieben

Instanziierung und Organisation

- Template Instanziierung beim g++
  - Cfront Model(AT&T Compiler)
    alle beim Übersetzen benötigten template-instanziierungen (bzw. deren definitionen) werden in einem REPOSITORY (ein Verzeichnis) gesammelt Ein Link-Wrapper übersetzt vorm linken alle notwendigen files im repository.

    Vorteil: man braucht keinen speziellen linker, optimale compile-time
    Nachteil: stark erhöhte Komplexität. Man kann nicht einfach mehrere Programme im gleichen Verzeichnis bauen oder ein Programm in mehreren Verzeichnissen!
    Folge: Trennung von template-definition und deklaration in separate files.

  g++ benutzt das Borland Model, kann aber auch mit repositories arbeiten oder die template instanziierung dem Benutzer überlassen.

Instanziierung und Organisation

- explizite Template Instanziierung und ihre Probleme
  Wir benutzen *explizite* Template Instanziierung.
  Option *-fno-implicit-templates* kein code für templates erzeugen und übersetzen. Man muss dann die templates explizit instanziieren:

```
Syntax: template volleDeklaration;
z.B.:
   template class matrix<int>;
   template void f<int>(matrix<int> &);
```

Instanziierung und Organisation

- Organisation des Quellcodes - oder: wann was wo einbinden?
  - Trennung von Deklaration(header), normaler Methoden Definition (cc-file) und template-Definitionen (tcc-file)
  - oder (wie bei uns z.Zt.:) Header enthalten auch template-definitionen aber: explizite Instanziierung in normalen files -> kein template-funktions-code
    + used_templates.cc file mit *IMPLIZITER* Instanziierung und *EXPLIZITEN Template-Definitionen*

    Vorteile: alle der expliziten Instanziierung (d.h. wenig Code-Duplizierung)
    + einfache Generierung von std-template-library template instanziierungen