Informatik II

Oliver Jack

Fachhochschule Jena
Fachbereich Elektrotechnik und Informationstechnik

Sommersemester 2010

Inhalt

Templates
- Template-Funktionen
- Template-Klassen
- Beispiel Binärer Suchbaum

Zusammenfassung

Vorlesung 9. Templates

Templates

Template-Funktionen

Template-Klassen

Beispiel Binärer Suchbaum

Zusammenfassung

9. Templates

Vorige Vorlesung

- Virtuelle Funktionen
- Abstrakte Klassen
- Mehrfachvererbung
- Virtuelle Vererbung
- Initialisierung bei abgeleiteten Klassen
- Konstruktionsschema für abstrakte Klassen

Heutige Vorlesung

- Parametrische Polymorphie
- Template-Funktionen
- Template-Klassen
- Beispiele Stack und binärer Suchbaum

Lernziele dieser Vorlesung

- Verständnis der Funktions- und Klassenspezialisierung
- Kenntnis des Template-Mechanismus in C++
- Kenntnis der Besonderheiten bei der Verwendung von Templates

Parametrische Polymorphie

- Die Benutzung des selben Programm-Codes für verschiedene Datentypen heißt parametrische Polymorphie.
- Der Datentyp ist ein Parameter des Programm-Codes.
- Parametrische Polymorphie wird zur Definition von Container-Klassen eingesetzt.
- Die Definition einer Funktion oder Klasse unter Verwendung parametrischer Polymorphie heißt Template.
- Templates ermöglichen Wiederverwendung von Programm-Code in einer typsicheren Weise.

Template-Funktionen

```
template <class klassenbezeichner[, class klassenbezeichner]*>
    typbezeichner funktionsbezeichner(parameterdeklarationen)
{
    anweisungen
}
```

- *klassenbezeichner* ist eine formaler Parameter für einen Datentyp, er wird innerhalb der Template-Definition verwendet
- Template-Definitionen erzeugen keinen Programm-Code
- Erst bei der Benutzung einer Template-Funktion wird Programm-Code erzeugt

Beispiel Maximum-Funktion

- Typunabhängige Maximum-Funktion
- TYP ist Parameter

```
template <class TYP> TYP maxi(TYP d1, TYP d2)
{
  if (d1 > d2) return (d1); else return (d2);
}
```

- Benutzung, als wäre eine Funktion für den Datentyp, den die Parameter besitzen, definiert
- Generierung der konkreten Funktion

```
float f = maxi (4.8, 7.9); // generiert
int i = maxi(100, 500); // generiert
char c = maxi('A', 'X'); // generiert
int j = maxi(600, 800); // nicht generiert
```

Funktionsweise. . .

- Erst bei Benutzung einer Template-Funktion wird Programm-Code erzeugt oder referenziert.
- Zunächst wird nachgeschaut, ob es eine konkrete Funktionsdefinition für die Datentypen der Parameter gibt, ist sie Vorhanden, so wird sie ausgeführt.
- Ist eine solche konkrete Funktion nicht vorhanden, so wird ein Template gesucht, aus dem sie generiert werden kann; ist ein solches Template vorhanden, so wird die konkrete Funktion generiert und dann ausgeführt.

...Funktionsweise

Quell-Code

```
template <class TYP>
TYP maxi(TYP d1, TYP d2) {
    if (d1>d2)
        return (d1);
    return (d2);
}

main ()
{
    float f=maxi(4.8, 7.9);
    int   i=maxi(100, 500);
    char ch=maxi('A", 'X');
    int   j=maxi(600, 800);
    ...
}
```

Generierter Code

```
float maxi(float d1, float d2) {
    if (d1>d2)
        return (d1);
    return (d2);
}

int maxi(int d1, int d2) {
    if (d1>d2)
        return (d1);
    return (d2);
}

char maxi(char d1, char d2) {
    if (d1>d2)
        return (d1);
    return (d2);
}

main ()
{
    float f=maxi(4.8, 7.9);
    int   i=maxi(100, 500);
    char ch=maxi('A", 'X');
    int   j=maxi(600, 800);
    ...
}
```

generiert

generiert

generiert

Funktionsspezialisierung

- Template-Funktionen realisieren implizit eine Art von Funktions-Overloading.
- Template-Funktionen können folgerichtig auch überschrieben werden.
- Das explizite Überschreiben von Template-Funktionen heißt Funktions-Spezialisierung.
- Aus der Funktionsweise von Template-Funktionen ergibt sich, dass bei Vorhandensein einer expliziten Funktionsdefinition das Template nicht benutzt wird.

Beispiel Maximum von Strings

- Die Template-Funktion `maxi()` ist nur brauchbar für Datentypen, auf denen der Vergleichsoperator > definiert ist.
- Für Strings muss eine andere Funktionsdefinition gegeben werden
- Funktions-Spezialisierung für Strings

```
char *maxi(char *d1, char *d2)
{
  if (strcmp (d1, d2) < 0 )
    return (d1);
  else
    return (d2);
}
```

Beispiel Maximum-Funktion mit Spezialisierung. . .

```
#include <iostream>

using namespace std;

template <class TYP> TYP maxi(TYP d1, TYP d2)
{
  if (d1>d2) return (d1); else return (d2);
}

// Spezialisierung fuer Strings
char *maxi(char *d1, char *d2)
{
  if (strcmp (d1, d2)<0)
    return (d1);
  else
    return (d2);
}
```

...Beispiel Maximum-Funktion mit Spezialisierung

```
int main ()
{
  cout << "maxi(1,␣2)␣=␣" << maxi(1, 2) << endl;
  // erzeugt aus Template

  cout << "maxi(2,␣1)␣=␣" << maxi(2, 1) << endl;
  // nicht erzeugt

  cout << "maxi(\"armin\",␣\"berta\")=␣␣"
       << maxi("armin", "berta") << endl;
  // nicht erzeugt (Spezialisierung)

  cout << "maxi(\"berta\",␣\"armin\")=␣␣"
       << maxi("berta", "armin") << endl;
  // nicht erzeugt (Spezialisierung)

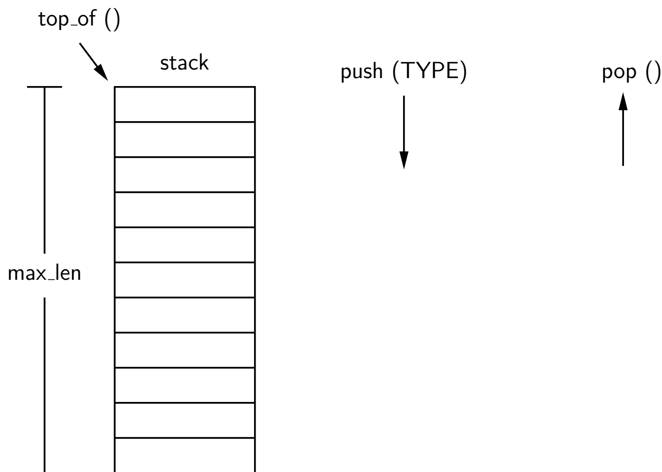
  return EXIT_SUCCESS;
}
```

Template-Klassen

```
template <class klassenbezeichner[, class klassenbezeichner]*>
    class klassenbezeichner {
    klasssendefinition
}
```

- Die Benutzung einer Template-Klasse erfordert die Angabe eines Datentyps für die formalen `class` *klassenbezeichner* (formale Parameter).
- Klassendefinitionen mit formalen Parametern heißen Container-Klassen.

Beispiel Stack. . .



...Beispiel Stack

- Ein Stack ist eine LIFO-Datenstruktur (Last-In-First-Out)
- Hauptfunktionen `push ()` (Datum einfügen) und `pop ()` (Datum entfernen)
- Nebenfunktionen `reset ()` (auf leeren Stack zurücksetzen), `top_of ()` (oberstes Datum), `empty ()` (Test auf leeren Stack), `full ()` (Test auf vollen Stack)
- Die Funktionen hängen logisch nicht von dem Datentyp der Daten ab
- Implementierung als Container-Klasse `stack<TYPE>`

Beispiel Stack stack.h...

```
// Template stack
template <class TYPE>
class stack {
private:
  enum { EMPTY = -1 };
  TYPE *s; // der Stack
  int max_len;
  int top;
public:
  stack (): max_len (1000)
    { s = new TYPE[1000]; top = EMPTY; }
  stack(int size): max_len(size)
    { s = new TYPE[size]; top = EMPTY; }
  ~stack () { delete [] s; }
```

...Beispiel Stack `stack.h`

```
// Stack-Methoden
  void reset () {
    top = EMPTY;
  }
  void push (TYPE c) {
    s[++top] = c;
  }
  TYPE pop () {
    return (s[top--]);
  }
  TYPE top_of () {
    return (s[top]);
  }
  bool empty () {
    return (bool(top == EMPTY));
  }
  bool full () {
    return (bool(top == max_len - 1));
  }
};
```

Beispiel Stack reverse.cpp

```
#include "stack.h"
#include <string>
using namespace std;

// Umdrehen eines Strings
string reverse(string str)
{
  int n(str.length());
  stack<char> stk(str.length());
  for (int i = 0; i < n; ++i) {
    stk.push(str[i]);
  }
  for (int i = 0; i < n; ++i) {
    str[i] = stk.pop();
  }
  return str;
}
```

Beispiel Stack `stackmain.cpp`

```
#include <iostream>
#include <string>
#include "stack.h"
using namespace std;

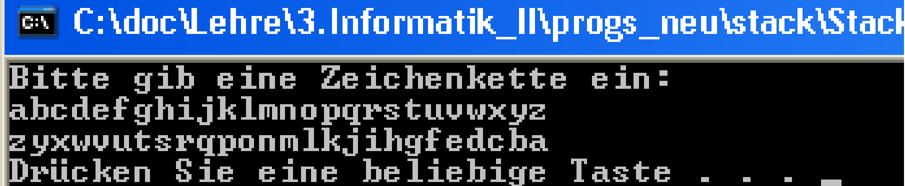
string reverse(string str);

int main ()
{
  string str;

  cout << "Bitte␣gib␣eine␣Zeichenkette␣ein:␣";
  cin >> str;
  cout << "umgekehrt:␣" << reverse(str) << endl;
  cout << "original:␣" << str << endl;
  return EXIT_SUCCESS;
}
```

Beispiel Stack: Benutzung

Programmausgabe



Klassen-Spezialisierung

- Wie bei Template-Funktionen wird eine explizite Definition zuerst benutzt bevor eine konkrete Definition aus einem Template erzeugt wird
- Für `stack<int>` werden die Funktionen `stack<int>::push()`, `stack<int>::pop ()`, etc. generiert
- Sollen Strings im Stack gespeichert werden (nicht Zeiger auf `char`), so kann `stack<char *>::push()` überschrieben werden (Klassen-Spezialisierung)

```
// s speichert string nicht Zeiger auf char
void stack<char *>::push (char *c)
{
  s[++top] = strdup (c);
}
```

Besonderheiten. . .

- Es können Nicht-Typen als Parameter verwendet werden; diese Parameter werden mit Konstanten instanziert
- Template-Funktionen können nur `class` Template-Parameter besitzen, und diese auch nur in der Parameterliste der Funktion
- ```
  template<class T, int n> class array_n {
  private:
    // n wird explizit angegeben
    T items[n];
  };

  // w ist 1000er Feld von complex
  array_n<complex, 1000> w;
  ```

```
template<class T> T foo () // illegal
{ ... }
```
```

...Besonderheiten...

Friend-Funktionen und Templates

- Eine Friend-Funktion, die keine Template-Spezifikation benutzt, ist Friend aller Instanzen der Klasse
- Eine Friend-Funktion, die eine Template-Spezifikation benutzt, ist Friend nur der konkret generierten Klasse

```
template <class T> class matrix {
private:
  // universell
  friend void foo_bar();
  // speziell
  friend vekt<T> produkt(vekt<T> v);
  // ...
};
```

. . . Besonderheiten

Static-Variablen und Templates

- Static-Member-Variablen sind nicht universell für alle konkret generierten Klassen, sondern immer nur speziell für einzelne konkret generierte Klassen.

```
template <class T>
class foo {
public:
  static int zaehler;
  // ...
};
foo<int> a;
foo<double> b;
```

- Die static Variablen `foo<int>::zaehler` und `foo<double>::zaehler` sind verschieden.

Templates und Includes

- Bei Benutzung von Templates funktioniert die übliche Quellcode-Organisiation mit Header-Dateien (`Datei.h`) und Definitionsdateien (`Datei.cpp`) nicht. Der Linker meldet Fehler.
- Abhilfe schafft hier, die gesamte Template-Klasse, d. h. Deklaration und Definition in eine Datei (`Datei.h`) zu packen.
- Alternativ kann am Ende von `Datei.h` die Datei `Datei.cpp` eingefügt werden (`#include "Datei.cpp"`). Dabei muss in `Datei.cpp` das einbinden von `Datei.h` entfernt werden (Löschen der Zeile `#include "Datei.h"`). Die Datei `Datei.cpp` muss dann vom Build ausgenommen werden.

Beispiel Binärer Suchbaum. . .

- Die Klasse `gen_tree` kann als Template-Klasse eleganter implementiert werden als mit generischen Zeigern und Vererbung.
- Der generische Zeiger `void *p_gen` wird hier zu einem Template Parameter `T`.
- Die Funktion `bnode<T>::print ()` ist hier nicht Friend-, sondern Member-Funktion.
- Die Klasse `gen_tree<T>` ist nicht für Vererbung vorgesehen; alle Hilfsfunktionen sind privat.

...Beispiel Binärer Suchbaum

- Die Vergleichsfunktion `int comp ()` ist nicht als Friend-Funktion definiert.
- Die Template-Funktion `int comp ()` setzt voraus, dass die Vergleichsoperatoren == und < auf dem Datentyp `T` definiert sind.
- Strings (Datentyp `char*`) werden durch Funktionsspezialisierung behandelt
- Die `print ()`-Funktion setzt voraus, dass der Operator << für den Datentyp T definiert ist. Die Template-Klasse `gen_tree<T>` erfordert nur Änderungen, die Objekte der Klasse `bnode` betreffen; Ersetzung durch `bnode<T>`.

Beispiel Binärer Suchbaum comp.cpp

```
template <class T>
int comp (T i, T j) // allgemeiner Fall
{
  if (i == j) // setzt voraus, dass ==, <
              // fuer T definiert sind
    return (0);
  else
    return ((i < j) ? -1 : 1);
}

int comp (char *i, char *j) // speziell fuer Strings
{
  return (strcmp (i, j));
}
```

Beispiel Binärer Suchbaum gentree.h...

```
// Template-Version generischer binaerer Suchbaeume
#include <iostream>
#include "comp.h"
using namespace std;
// forward-Deklaration
template<class T> class gen_tree;

template<class T> class bnode {
private:
  friend class gen_tree<T>;
  bnode<T> *left;
  bnode<T> *right;
  T data;
  int count;
  bnode (T d, bnode<T> *l, bnode<T> *r)
    : left (l), right (r), data (d), count (1) {}
```

...Beispiel Binärer Suchbaum gentree.h...

```
  void print ()
  {
  // setzt voraus, dass << fuer T definiert ist
    cout << data << " (" << count << ")\t";
  }
};
```

...Beispiel Binärer Suchbaum gentree.h...

```
template <class T> class gen_tree {
private:
  bnode<T> *root;
  T find (bnode<T> *r, T d);
  void print (bnode<T> *r);
public:
  gen_tree () { root = 0; }
  void insert (T d);
  T find (T d)
  {
    return (find (root, d));
  }
  void print () { print (root); }
};
```

Beispiel Binärer Suchbaum gentree.cpp...

```
template <class T>
void gen_tree<T>::insert (T d)
{
  bnode<T> *temp = root;
  bnode<T> *old;

  if (root == 0) {
    root = new bnode<T>(d, 0, 0);
    return;
  }
  while (temp != 0) {
    old = temp;
    if (comp (temp->data, d) == 0) {
      (temp->count)++;
      return;
    }
```

...Beispiel Binärer Suchbaum gentree.cpp...

```
    if(comp(temp->data, d) > 0)
      temp = temp->left;
    else
      temp = temp->right;
  }
  if (comp (old->data, d) > 0)
    old->left = new bnode<T>(d, 0, 0);
  else
    old->right = new bnode<T>(d, 0, 0);
}
```

...Beispiel Binärer Suchbaum gentree.cpp...

```
template <class T>
T gen_tree<T>::find (bnode<T> *r, T d)
{
  if (r == 0)
    return (0);
  else if (comp (r->data, d) == 0)
    return (r->data);
  else if (comp (r->data, d) > 0)
    return (find (r->left, d));
  else
    return (find (r->right, d));
}
```

...Beispiel Binärer Suchbaum gentree.cpp

```
template <class T>
void gen_tree<T>::print (bnode<T> *r)
{
  if (r != 0) {
    print (r->left);
    r->bnode<T>::print ();
    print (r->right);
  }
}
```

Beispiel Binärer Suchbaum main.cpp...

```
#include <iostream>
#include <string.h>
#include "gentree.h"
using namespace std;
int main ()
{
  char dat[256];
  gen_tree<char*> t;
  char *p;
  int item;
  cout << "Bitte␣Zeichenketten␣eingeben:" << endl;
  while (cin >> dat && cin.good ()) {
    p = new char[strlen (dat)+1];
    strcpy (p, dat);
    t.insert (p);
  }
  t.print ();
  cout << endl << endl;
```

...Beispiel Binärer Suchbaum main.cpp

```
  // Baum mit Zufallszahlen fuellen
  // Initialisierung des Generators
  srand(unsigned (time(NULL)));
  gen_tree<int> i_tree;
  for (int i=0; i<15; i++) {
    item = rand ();
    cout << item << "␣␣";
    i_tree.insert (item);
  }
  cout << endl;
  i_tree.print ();
  cout << endl;
  return (0);
}
```

Beispiel Binärer Suchbaum: Benutzung

Programmausgabe

C:\doc\Lehre\3.Informatik_II\progs_neu\btreepar\btreepar.exe

```
Bitte Zeichenketten eingeben:
Hans
Alfred
Karl
Wolfgang
Hans
Ilse
Margot
^Z
Alfred (1)      Hans (2)        Ilse (1)        Karl (1)        Margot (1)
Wolfgang (1)

21226  25884  12469  1867  22419  10769  27209  15738  6024  8426  18134  13519
 6619  13122  19737
1867 (1)        6024 (1)        6619 (1)        8426 (1)        10769 (1)
12469 (1)       13122 (1)       13519 (1)       15738 (1)       18134 (1)
19737 (1)       21226 (1)       22419 (1)       25884 (1)       27209 (1)
```

Templates

- In C++ können typunabhängige Algorithmen mit Hilfe von Templates implementiert werden.
- Klassen können mit Templates als Container-Klassen realisiert werden.
- Besonderheiten bei der Verwendung von Templates gibt es bei Friend-Funktionen und statischen Variablen.