Grundlagen der Informatik (Teil II - Ergänzungen)

Dr. Andreas Müller, TU Chemnitz, Fakultät für Informatik

Inhaltsverzeichnis

An der Entstehung dieses Kursmaterials waren beteiligt:
Heino Gutschmidt
Uwe Hübner
Thomas Müller
Holger Trapp

6 Objektorientierte Programmierung - zusätzliche Kapitel

6.1 Statische Member

- **Klassenvariablen (statische Membervariablen)**

Instanzvariablen	Klassenvariablen
jedes Objekt hat eigene Instanzvariablen	mit allen Objekten seiner Klasse geteilt
individuelle Werte je Objekt	für jedes Objekt identische Werte (je Klasse nur einmal vorhanden)
nur über Objekte zugreifbar	zugreifbar, ohne daß ein Objekt existieren muß

```
class A {
     static int a;    // private Klassenvariable
public:
     static int b;    // oeffentliche Klassenvariable
     void meth();
};

void A::meth()
{
     a = ... ;
}

int A::a = 1;              // Definition
int A::b;

void f()
{
     A object;
```

```
    object.b = 2;    // Zugriff ueber Objekt
    A::b = 3;        // Zugriff ueber die Klasse
}
```

- **Klassenmethoden (statische Memberfunktionen)**

Methoden	Klassenmethoden
Botschaft an Objekt (nur für Objekte aufrufbar)	aufrufbar, ohne, daß ein Objekt existieren muß
Zugriff auf Instanz- und Klassenvariablen	Zugriff nur auf Klassenvariablen

```
class A {
     ...
public:
     static void meth();     // oeffentliche Klassenmethode
};

void A::meth()
{
     ...
}

void f()
{
     A object;

     A::meth();              // Aufruf ueber die Klasse
     object.meth();          // Aufruf ueber Objekt
}
```

6.2 Templates und Exception Handling

- **Templates werden genutzt, um generische Funktionen und Klassen zu verwenden.**
- **In einer generischen Funktion oder Klasse wird der Datentyp verwendet, der dieser Funktion als Parameter übergeben wurde.**
- **Exception Handling ist ein C++-Subsystem, welches es erlaubt, Laufzeitfehler in strukturierter und gesteuerter Art und Weise zu behandeln. (kein unkontrollierter Absturz)**
- **Der prinzipielle Vorteil besteht darin, das Codieren des Error Handlings, welches bisher durch Hand in jedem größeren Programm eingefügt wurde, zu automatisieren.**
- **Sowohl Templates als auch das Exception Handling sind nicht Teil des ursprünglichen C++-Entwurfs, sind aber im ANSI-C++-Standard enthalten (Achtung: nicht alle Compiler beherrschen beide Features).**

6.2.1 Generische Funktionen

```
 template<class Ttype>return-type function-name(parameter-List)
{
//body of function
}
```

- **Formulierung des Algorithmus unabhängig vom Datentyp**
- **Compiler bestimmt automatisch den Datentyp und generiert den dafür korrekten Code (d.h. man entwirft eine Funktion, die sich automatisch überlädt.).**
- **Es können mehr als ein Typ class Ttype angegeben werden (TType ist Platzhalter für den konkreten Typ).**

- **Unterschied zu überladenen Funktionen: mehr restriktiv (es ist nicht möglich, eine andere Funktionalität zu bestimmen).**
- **Es ist möglich, eine generische Funktion mit dem Namen X und eine "herkömmliche" Funktion mit dem gleichen Namen anzugeben. Dann muß die Parameterliste unterschiedlich sein.**

Beispiel:

```
#include <iostream.h>

template <class X> void swap(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}

int main()
{
int i = 10, j = 20;
float x = 10.5, y = 20.9;

cout << i << " " << j << endl;
cout << x << " " << y << endl;
swap(i,j);
swap(x,y);
   cout << i << " " << j << endl;
cout << x << " " << y << endl;
return 0;
}
```

6.2.2 Generische Klassen

```
template <class Ttype> class class-name
{
   .....
}
```

- **Definition der Algorithmen, die durch die Klasse benutzt werden.**
- **Datentyp wird als Parameter bei der Instanziierung der Obkjekte dieser Klasse spezifiziert.**
- **Nützlich, wenn generalisierbare Algorithmen benutzt werden.**
  Beispiel: Die Verwaltung einer einfach verketteten Liste ist unabhängig vom Typ der Datenelemente der Liste.
- **Der Compiler generiert bei der Instanziierung den korrekten Typ der Objekte.**

Beispiel:

```
#include <iostream.h>

template <class data_t> class list
{
data_t data;
list *next;
  public:
list (data_t d);
void add (list *node)
  {node->next = this; next = 0;}
list *getnext()
  {return next;}
data_t getdata()
  {return data;}
};
```

```
template <class data_t> list>data_t>::list(data_t d)
{
data = d;
next = 0;
}

main()
{
list<char> start('a');
list<char> *p, *last;
int i;

last = &start;
for (i=1; i < 26; i++)
{
p = new list<char> ('a'+i);
p->add(last);
last = p;
}
p = &start;
while(p)
{
cout << p ->getdata();
p = p->getnext;
 }
return 0;
}
```

6.2.3 Exception Handling

```
try
{
 // try block
 throw exception1 type1
 throw exception2 type1
 throw exception1 type2
 ....
}
catch (type1 arg)
{
 // catch block
}
catch (type2 arg)
{
 // catch block
}
...
```

- **C++ bietet einen eingebauten Mechanismus zur Fehlerbehandlung: das Exception Handling**
- **Schlüsselwörter try, throw, catch**
- **Wenn eine Exception (z.B. ein Fehler) in einem try-Block auftritt, wird sie durch throw an die Umgebung des try-Blocks gesendet. Der catch-Block nimmt die Exception auf und verarbeitet sie.**
- **Wenn zu einer Exception kein Catch-Block existiert (nach den Regeln des Argument-Matchings), kann ein unnormales Programmende folgen.**

```
#include <iostream.h>

main()
{
cout << "start \n";

try
{
cout << "inside try block \n";
int j,k,l;
cin >> j >> k;
if (k == 0)
throw 50;
 else
l = j / k;
cout << l << endl;
 }

catch (int i)
{
cout << "Caught One! number is: ";
cout << i << endl;
}

cout << "end \n";
return 0;
}
```