

Zeiger (Pointer), Referenzen und const

Tobias Sielhorst

10 May 2006

chair for computer aided medical procedures

department of computer science | technische universität münchen

Typen

Ein Zeiger (MyObject*)

- zeigt auf einen Speicherbereich
- Es gibt Nullzeiger (Nullpointer)

Eine Referenz (MyObject&)

- ist wie ein neuer Name für ein bestehendes Objekt
- Es gibt keine Nullreferenz

JAVA kennt nur die Mischung aus beiden

```
MyObject object;  
MyObject* objectPointer;  
  
MyObject& myObjectReference  
           = object;
```

Umwandlung

- Die gleichen Zeichen werden bei Umwandlungen benutzt
- & vor einem Objekt bedeutet
“Ich hätte gern den Speicherort vom Objekt”
- * vor einem Zeiger (bzw. Speicherort) bedeutet
„Ich hätte gern die Referenz (bzw. das Objekt) zum Zeiger“

```
MyObject object;  
MyObject* objectPointer;  
  
objectPointer = &object;  
  
MyObject& objectReference=  
*objectPointer;
```

Wozu das ganze? Geschwindigkeit!

- Objekte beliebige Größe
- Pointer → 4 Byte
Referenzen → 4 Byte

```
Telefonbuch t;  
string suchbegriff;  
Eintrag e;  
e = suchfunktion(suchbegriff, t);
```

```
Eintrag suchfunktion(string s, Telefonbuch t)  
{ // kopiert beim Aufruf das ganze Telefonbuch  
  ...  
  return e;  
}
```

```
Eintrag suchfunktion(string& s, Telefonbuch& t)  
{ // kopiert nur 4 Byte  
  ...  
  return e;  
}
```

```
Eintrag& suchfunktion(string& s, Telefonbuch& t)  
{ // wenn man den Eintrag ändert, ändert sich  
  // auch der im Telefonbuch!  
  ...  
  return e;  
}
```

Warum also Zeiger?

```
MyParentClass* mp;  
MyClass mc;  
  
mp = &mc; // geht  
  
MyParentClass& mr = mc; //Kompilierfehler
```

- In vielen Fällen möchte man ein Objekt über eine höhere Vererbungs-klasse (parent class) ansprechen. Dies geht nur mit Zeigern
- Effizientes Programmieren mittels Zeigerarithmetik

```
// ein Byte Array (10MByte) um den Wert 5 erhöhen  
char* array = new char[10000000];  
for (int i=0; i <10000000;i++){  
    array[i]+=5;  
}  
delete[] array;
```

- folgendes Programm ist x-mal schneller

```
char* array = new char[10000000];  
char* endOfArray = array+10000000;  
for (char* i=array; i <endOfArray;i++){  
    (*i)+=5;  
}  
delete[] array;
```

Der Keller und die Halde (Stack und Heap)

- Es gibt zwei Möglichkeiten wo man Speicher für seine Objekte bekommt:

```
MyObject mo1; // Stack  
MyObject* mo2 = new MyObject(); // Heap
```

- Beim Stack wird der Speicher wieder überschrieben, nachdem die Variable aus dem lokalen Namensraum verschwindet
- Beim Heap wird der Speicher erst wieder freigegeben, wenn man die Funktion `delete` aufruft
- Bei JAVA kommt alles auf den Heap und wenn es nicht mehr gebraucht wird vom GarbageCollector wieder freigegeben



**Fehlerquelle
Nr. 1 !**

Tipps zu Zeigern (Pointer) und Referenzen

- Man nimmt Pointer, wenn man Vererbungsverhältnisse zur Laufzeit ausnutzen möchte
- Man nimmt Pointer, wenn man ausdrücken möchte, dass eventuell der Nullpointer in der Variable ist
- Man nimmt Referenzen, wenn man sich absolut sicher ist, dass das Objekt vorhanden ist
- Wenn man `new` schreibt, sollte man sich angewöhnen auch gleich das `delete` dazu zu schreiben
- Man sollte sich angewöhnen, alle Pointer mit 0 zu initialisieren und vor `delete` auf 0 zu überprüfen
- Zu `new MyObject[42]` gehört `delete []`
- Destruktoren sollten immer virtuell sein (`virtual ~MyObject` im Header)

Wie sage ich es meinem Kollegen? Mit const deklarieren

- Ich biete eine Funktion an, bei der die Übergabeobjekte nicht verändert werden

```
Eintrag& meineFunktion(const Telefonbuch& t, const string& s);
```

- Ich biete eine Funktion an, bei dem keine Membervariablen (auch nicht indirekt) geändert werden

```
Eintrag& meineFunktion(meinObject& mo1, meinObject& mo2) const;
```

- Ich biete eine Funktion an, bei dem der Übergabe nicht geändert werden darf, damit nicht schlimmeres passiert

```
const Eintrag& meineFunktion(meinObject& mo1, meinObject& mo2);
```

Tipps zu const

- Der Compiler prüft, ob der Code zu den `const` Deklarationen korrekt ist und wirft einen Fehler (keine Warnung) falls nicht
- Man muss nicht mit `const` programmieren, aber es hilft
- Mit einem Cast kann man dem Compiler sagen, dass man doch auf eine konstante Variable zugreifen möchte
- `const` immer von rechts nach links lesen!
 - `const MyObject*` ist ein Pointer auf ein konstantes Objekt
 - `MyObject const *` ist ein konstanter Pointer auf ein Objekt
 - `const MyObject const *` ist eine konstanter Pointer auf ein konstantes Objekt

Klassen, Namensräume (namespaces) und Templates

Tobias Sielhorst

10 May 2006

chair for computer aided medical procedures

department of computer science | technische universität münchen

Namensraum (namespace)

- Bei größeren Projekten wird die Namensgebung von Klassen und Funktionen unübersichtlich
- Deshalb wurden in C++ Namensräume eingeführt

```
namespace myNamespace{  
    class MyClass{  
        MyClass();  
        void myNonsenseFunction(){ return; }  
    };  
}
```

myClass.h

```
include "myClass.h"  
int main(){  
    myNamespace::MyClass someObject;  
    someObject = new myNamespace::MyClass;  
    return 0;  
}
```

main.cpp

```
include "myClass.h"  
using namespace myNamespace;  
int main(){  
    MyClass someObject;  
    someObject = new MyClass;  
    return 0;  
}
```

main.cpp

Beispiel: Die Klasse string in std

- Die Klasse string bietet einen wesentlich komfortableren Zugriff auf Zeichenketten als char*

```
include <string>
int main(){
std::string s = "Was ist die Losung";
std::string t;
std::cout << s << std::endl;
std::cin >> t;
if( t.compare("per aspera ad astra")==0 ){
    std::cout << "Richtig!"<< std::endl;
}
return 0;
}
```

Klassen und Objekte

- Kurz zur Wiederholung
 - Klasse ist ein Objekttyp --- Objekt ist eine Instanz einer Klasse
 - Objekte bestehen aus Daten (Membervariablen) und zugehörigen Funktionen (Memberfunktionen)
 - statische Variablen gibt es von einer Klasse nur *einmal* pro Prozess
 - eine Änderung einer statischen Variable wirkt sich auf *alle* Instanzen aus
 - statische Funktion können ohne Instanz aufgerufen werden (dürfen aber natürlich nicht auf Member zugreifen)
 - nur virtuelle Funktionen in vererbten Klassen können überschrieben werden und werden zur Laufzeit bestimmt
 - abstrakte Klassen sind Klassen, von denen man kein Objekt erzeugen kann, weil nicht alle virtuellen Funktionen implementiert sind
 - Pointer auf abstrakte Klassen sind sehr wohl möglich (und sinnvoll)

Klassen und Objekte: Der Header

```
class MyClass:MyParentClass
{ // die Klasse MyClass erbt von MyParentClass
public:
    MyClass(); //Konstruktor
    MyClass(std::string text); //zweiter Konstruktor
    virtual ~MyClass; // Destruktor

    void fu(); // eine Funktion
    virtual int fun(); // eine virtuelle Funktion
    virtual int func()=0; // eine rein virtuelle (=abstrakte) Funktion
    static double funct(); // eine statische Funktion

    std::string m_someString; // eine Membervariable
    static int m_someNumber; // eine statische Membervariable
}
```

myClass.h

Klassen und Objekte: Die Implementierung

```
include "myClass.h"
```

```
myClass.cpp
```

```
MyClass::m_someNumber = 5;
```

```
MyClass::MyClass(){  
    m_someString = "Eine Intialisierung vom Text";  
}; //Konstruktor
```

```
MyClass::MyClass(std::string text){  
    m_someString = text;  
}; //zweiter Konstruktor
```

```
MyClass::~MyClass(){}; // Destruktor
```

```
void MyClass::fu(){}; // eine Funktion
```

```
int MyClass::fun(){return 4;}; // eine virtuelle Funktion
```

```
double MyClass::funct(){return 2;}; // eine statische Funktion
```

Gegenüberstellung Syntax

C++

- Vererbung :
- Mehrfachvererbung
:Klasse1, Klasse2
- Zugriff auf Member
objekt.funktion() bei
Referenzen und
objekt->funktion() bei
Pointern
- globale Variablen und
Funktionen
- Templates <>

Java

- Vererbung `extends`
- Mehrfachvererbung
`implements Klasse1, Klasse2`
(nur für abstrakte Klassen
ohne Membervariablen)
- Zugriff auf Member
`objekt.funktion()`
- keine globale Variablen
- Generics (ab JAVA 1.5)

Typenunabhängige Programmierung mit Templates

- Mit Templates kann man Programme vom Datentyp abstrahieren aber schon zur Compilezeit prüfen lassen.
- Der Code ist exakt genau so schnell wie Code ohne Templates
- Beispiel: STL
(Standard Template Library)
- [mehr zur STL](http://www.cosy.sbg.ac.at/~held/teaching/wiss_arbeiten/slides_01-02/fimberger.pdf)
(http://www.cosy.sbg.ac.at/~held/teaching/wiss_arbeiten/slides_01-02/fimberger.pdf)
- [auf Englisch und ausführlich](http://www.sgi.com/tech/stl/)
(<http://www.sgi.com/tech/stl/>)

```
include <list>
using namespace std;
int main(){
    list<int> zahlenReihe;
    list.push_back(4);
    list.push_back(2);

    list<int>::iterator it;
    it = list.begin();
    cout << (*it) << endl;
    it++;
    cout << (*it) << endl;
    return 0;
}
```

Debuggen in C++ anhand des integrierten Debuggers im VS

Tobias Sielhorst

09 May 2006

chair for computer aided medical procedures

department of computer science | technische universität münchen

Integrierte Debugger

- Hier: Debugger des Visual Studio 2005
- Kompilieren im Debugmodus:
 - Der Compiler fügt zusätzliche Informationen dem Code hinzu, die das Programm semantisch nicht verändern
 - Damit kann man kann das Programm Schritt für Schritt im Programmtext ausführen (obwohl das Programm nach wie vor im Binärcode läuft!)
 - Es finden keinen zusätzlichen Optimierungen mehr statt

Haltepunkte (Breakpoints)

- Das Programm hält immer am Haltepunkt, wenn es im Debugmodus kompiliert wurde UND im Debugmodus gestartet wurde (F5)
- Unter „Haltepunkteigenschaften bearbeiten“ kann man unten im Dialog Bedingungen an den Haltepunkt knüpfen, ohne den Code zu ändern
- Es gibt mehrere Fenster in dem man Variablen ansehen (und auch verändern!) kann
 - Beobachten: Hier kann man auf der linken Seite Variablen eintragen
 - Auto: Hier sind automatisch Vorschläge für Variablen enthalten
 - Natürlich sind nur Variablen möglich, die an dieser Stelle im Code bekannt sind!

Aufrufliste (Call stack)

- In der Aufrufliste kann man sehen, von wo die Funktion in der der Haltepunkt ist, aufgerufen wurde
- Mit einem Doppelklick kann man in die Funktion springen, die aufgerufen hat – dann kann man sich auch die Variablen ansehen, die in der Funktion bekannt waren

Ändern und Kompilieren (Edit and Compile)

- Das Visual Studio bietet die Möglichkeit, wenn das Programm unterbrochen wurde, eine Änderung vorzunehmen, zu kompilieren und *an der unterbrochenen Stelle* fortzufahren!
- Das ist aus technischen Gründen nicht immer möglich, aber kann sehr hilfreich sein

Ein Schritt zurück...

- ... wäre der Traum aller Programmierer, aber ist technisch nicht korrekt realisierbar
- Wenn man den gelben Pfeil im Debugger hochzieht, wird *nicht* der Programmlauf (mit Variablen) zurückgespult, sondern das Programm führt als nächsten Befehl den Befehl nach der Pfeilspitze aus