 © 2001 Markus Rigling

Programmieren:

Smart Pointer

1. Auflage

Inhaltsverzeichnis:

1. Smart Pointer....3
2. Vor- Nachteile von Smart Pointer....3
3. auto_ptr....3
4. Umsetzung / Anwendung: 4
5. Wertzuweisung / Kopieren eines auto_ptr....5
6. Dereferenzierung....6
7. Test auf NULL....7
8. Typkonversion....7

1. Smart Pointer

Smart Pointer sind Objekte, die wie eingebaute Zeige3r aussehen, sich so verhalten und so benutzt werden können. Es kann unterschiedliche Smartpointer in unterschiedlichen Einsatzgebieten geben.

Ein Smart Pointer kann folgende zusätzliche Funktionalität bieten:

- Da ein Smart Pointer ein Objekt ist kann er mit Hilfe seines Konstruktors / Destruktors das Verhalten beim Erzeugen und Vernichten festgelegt werden.
- Er kann initialisiert werden
- Über Kopierkonstruktor und Wertzuweisungsoperator kann das Verhalten bei diesen Operationen festgelegt werden.
- Individuelle Anpassung bei Dereferenzieren

2. Vor- Nachteile von Smart Pointer

- grössere Sicherheit gegenüber Speicherlecks
- nicht völlig gleiches Verhalten wie eingebaute Zeiger
- Keine Vererbungserkennung (ausser ab C++ 3.0)
- Testen und Debuggen ist etwas komplizierter
-

3. auto_ptr

siehe auch im Script Seite 197 – 199

`auto_ptr` ist ein einfacher Smart Pointer mit der Funktion, das Memory, auf das er zeigt, beim Vernichten freizugeben. Er löst diese Aufgabe sicher und auf einfache Art. Seine Verwendbarkeit ist auf Grund seiner Einfachheit allerdings beschränkt.
`auto_ptr` ist Teil der C++ Standardbibliothek, liegt also fertig implementiert vor.

Deklaration:

```
template <class Type>
class auto_ptr {
public:
     auto_ptr( Type* ptr = 0 );
     ~auto-ptr();
     ...
private:
```

```
    Type* pointee;      //Attribut ist Dumb Pointer auf Type
};
```

Definition:

```
template <class Type>   //Konstruktor initialisiert pointee auf 0
auto_ptr<Type>::auto_ptr( Type* ptr )
    : pointee( ptr )
{}

template <class Type>   //Destruktor vernichtet Objekt, wo
                        //pointee zeigt
auto_ptr<Type>::~auto_ptr()
{
    delete pointee;
}
```

4. Umsetzung / Anwendung:

```
void main()
{
    auto_ptr<Kreis> apk = new Kreis(...);
    ...
}                       //auto_ptr vernichtet Kreis
```

Hier wird `auto_ptr` mit der Klasse `Kreis` instantiiert und das `auto_ptr`-Objekt `apk` dynamisch (von Klasse Kreis) erzeugt.

Man darf den Smart Pointer nur mit dynamisch angelegten Objekten verwenden! Ausserdem muss man beachten, dass das Objekt dem Smart Pointer gehört.
Ausserdem ist auch folgendes Beispiel fehlerhaft:

```
{
    Kreis* pk = new Kreis(...);   //dynamisch angelegter Kreis
    auto_ptr<Kreis> apk = pk;
    ...
    delete pk;                    //FALSCHE Verwendung, der
                                  //Kreis gehört apk, pk
                                  //sollte nicht mehr
                                  //verwendet werden!
}
```

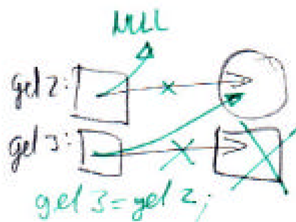
Ausserdem gehört das Objekt, auf das der Smart Pointer zeigt, nur noch dem Smart Pointer und sollte möglichst nur noch über diesen benutzt werden. Auf keinen Fall darf das Objekt von Hand vernichtet werden.

Im folgenden Beispiel wird ein Smart Pointer korrekt angewendet (im Zusammenhang mit Vererbung). Er zeigt auf Grafikelement und wird mit einem dynamisch angelegten Objekt der

Klasse `Kreis` initialisiert. Dies ist möglich, da ein `Kreis` eine Spezialisierung von `Grafikelement` ist, die Adresse von `Kreis` (Ergebnis von `new Kreis`) kann als Adresse eines Grafikelements an den Konstruktor übergeben werden.

```
{
      auto_ptr<Grafikelement> gel = new Kreis(...);
      ...
}                                //~auto_ptr vernichtet Kreis
```

5. Wertzuweisung / Kopieren eines auto_ptr




```
auto_ptr<Grafikelement> gel1 = new Kreis(...);
auto_ptr<Grafikelement> gel2 = gel1;    //gel1 zeigt nicht mehr
                                        //auf Kreis
auto_ptr<Grafikelement> gel3;
gel3 = gel2;                            //gel2 zeigt nicht mehr
                                        //auf Kreis
}                                       //Kreis wird durch gel3
                                        //vernichtet
```

Folge 1:
Man macht keine Kopien oder Wertzuweisungen von `auto_ptr`'s in einem Block, wie in obigem Beispiel. Andernfalls sollte man einen anderen Smart Pointer verwenden.

Folge 2:
Nie ein `auto_ptr` als Parameter einer Funktion übergeben.

Folge 3:
Das geht problemlos:

```
{
      auto_ptr<Kreis> apk;
      apk = auto_ptr<Kreis>( new Kreis(...) );
}
```

Folge 4:
Problemlos ist auch die Rückgabe von `auto_ptr` aus Funktionen.

```
auto_ptr<Kreis>
```

```
foo( )
{
     auto_ptr<Kreis> p = new Kreis(...);
     return(p);
}                         //p wird gelöscht, zeigt auf NULL
```

p wird an z.B. apk zurückgegeben `apk = foo();` p wird beim Verlassen der Funktion vernichtet. Das ist aber kein Problem, da `p` nicht mehr Eigentümer von `new Kreis` ist.

6. Dereferenzierung

`apk` sei ein `auto_ptr`. Durch den Aufruf `apk->zeichnen();` oder `(*apk).zeichnen();` ist es möglich, auf das Objekt selbst zuzugreifen. Der `operator*()` und `operator->()` sind im Template `auto_ptr` also auch überladen:

Deklaration:

```
public:
     ...
     Type& operator*() const;
     Type* operator->() const;
};
```

Definition:

```
template <class Type>
Type& auto_ptr<Type>::operator*() const
{
     return( *pointee );
}

Type* auto_ptr<Type>::operator->() const
{
     return( pointee );
}
```

7. Test auf NULL

Soll ein Smart Pointer bei der Erstellung auf NULL initialisiert werden, ist dies mit dem `operator void*()` zu realisieren (Typkonversion).

Deklaration:

```
operator void*() const;
```

Definition:

```
template <class Type>
auto_ptr<Type>::operator void*() const
{
     return( (void*)pointee );
}
```

oder

```
template <class Type>
auto_ptr<Type>::operator void*() const
{
     return( (void*)(pointee !=0) );
}
```

In der ersten Implementierung wird der Zeiger `pointee` direkt zurückgegeben und könnte vom Aufrufer benutzt werden. In der zweiten Implementierung wird `0` oder `1` nach `void*` gewandelt und zurückgegeben.
Diese Lösung hat auch eine unschöne Konsequenz: der Vergleich `apk == apr` funktioniert, indem beide Objekte nach `void*` gewandelt werden.

```
auto_ptr<Kreis> apk = new Kreis(...);
auto_ptr<Rechteck> apr = new Rechteck(...);
...
if( apk == apr ) { ... }
```

8. Typkonversion

Manchmal benötigt eine bereits existierende Software in ihren Funktionen und Methoden einen eingebauten Zeiger. Um diese Funktionen benutzen zu können, muss man den Smart Pointer in einen eingebauten Zeiger umwandeln können. Das kann man auf zwei Arten machen:

```
template <class Type>
class auto_ptr {
public:
     ...
```

```
     operator Type*() const;        //Typumwandlungsoperator
     Type* get() const;             //gebt auch pointee zurück
};

template yclass Type>
auto_ptr<Type>::operator Type*() const
{
     return( pointee );
}

template <class Type>
Type* auto_ptr<Type>::get() const
{
     return( pointee );
}
```

Der Typumwandlungsoperator ist in der Benutzung einfach, dafür können ungewollte Typwandlungen auftreten. Die Methode get() muss man explizit benutzen, sie ist daher sicherer.

```
auto_ptr<Kreis> apk = new Kreis( ... );
auto_ptr<Rechteck> apr = new Rechteck( ... );
foo( apk );          //Aufruf von apk.operator Kreis*
goo( apr );          //Aufruf von apk.operator Rechteck*
Kreis* pk;
pk = apk;            //Möglicherweise ungewollte Typwandlung
foo( apk.get() );    //expliziter Aufruf von apk.get
goo( apr.get() );    //expliziter Aufruf von apr.get
Kreis* pk;
pk = apk.get();      //explizit gewollte Typwandlung
```

Beide Methoden beinhalten die Gefahr des Missbrauchs: man kann aus einem Smart Pointer einen Dumb Pointer machen. Und sobald man diesen Dumb Pointer benutzt, wird der Smart Pointer umgangen und sein schlaues Verhalten nicht mehr voll genutzt.

Die Vererbungsstruktur wird bei Smart Pointern auch nicht gekannt. So ist es mit Smart Pointer nicht möglich, was anhand von Dumb Pointern problemlos realisiert werden kann:

```
Grafikelement* ge;
Kreis* pk = new Kreis(...);
ge = pk;             //Typwandlung in Oberklasse
const Kreis* cpk;    //Zeiger auf konstante kreise
cpk = pk;            //cpk zeigt auf pk, aber als konstanter
Kreis
```

Mit unserem Pointer geht das (wie schon gesagt) nicht:

```
auto_ptr<Grafikelement> age;
auto_ptr<Kreis> apk = new Kreis(...);
age = apk;           //COMPILERFEHLER!!!!!
```