



Besser programmieren

Erfolgreich Klippen umschiffen

Texteditor angeworfen, Programm heruntergeladet, übersetzt, fertig. So einfach funktioniert das nicht, wenn in C++ entwickelt wird. Eines der Hauptprobleme ist dabei C++ selbst. Die Sprache bietet nicht nur eine Unmenge an Sprachfeatures, sondern mindestens ebenso viele Fallstricke.

THOMAS WÖLFER

Guter Rat



#define vermeiden

Dies ist einer der häufigsten Ratschläge, die C-Programmierer von C++-Programmierern erhalten. Es gibt jede Men-

ge guter Gründe weshalb man #define vermeiden sollte, letztendlich kommen alle auf den gleichen Hauptgrund zurück: #define ist nicht direkt ein Teil von C++ (oder von C) – zumindest nicht, was den Compiler betrifft, denn der C/C++-Compiler sieht ein per #define definiertes Symbol nie unter seinem symbolischen Namen. Hierzu ein Beispiel: Angenommen man verwendet das Präprozessor-Statement

```
#define PI 4.0
```

in einem Header-File, um später in den C/C++-Dateien das Symbol PI verwenden zu können. Das ist gut und schön – und man kann das auch tun – nur leider sieht der Compiler das Symbol PI nie unter diesem Namen. Statt dessen sieht er nur den Wert "4.0", denn das ist das, was im Quelltext nach dem Präprozessorlauf vor dem Kompilieren übrig bleibt. Wenn man sich innerhalb des Debuggers befindet, gibt es ein Problem. Auch der Debugger kennt das Symbol "PI" nicht, denn schließlich wurde dieses zur Übersetzungszeit nie verwendet.

Hier noch ein zweites Beispiel für eine häufige Benutzung des Präprozessors, die man in C++ besser anders erstellt:

```
#define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Hier handelt es sich um eine typisches "MAX"-Makro, mit dem der maximale von zwei Werten ermittelt werden kann. Praktisch an diesem Makro: Man kann es mit jedem beliebigen Datentyp füttern, es funktioniert immer – zumindest dann, wenn identische Datentypen verwendet werden.

Dummerweise handelt es sich hier aber um ein Makro. Das bedeutet: Der Compiler kommt nicht erneut zum Einsatz, sondern der Präprozessor, und das mit un schönen Folgen. Schreibt man unter Verwendung der angegebenen Makros die folgenden Ausdrücke:

```
int a = 1;
int b = 0;
int c;
c = MAX( a++, b );
```

sieht der Compiler – weil es sich bei MAX() um ein Makro handelt – zur Übersetzungszeit das folgende Statement:

```
c = ((a++) > (b) ? (a++) ? (b) )
```

Eine unangenehme Sache, denn "a" wird hier gleich zweimal inkrementiert. Das ist mit Sicherheit nicht, was der Programmierer eigentlich beabsichtigte. Besser umgeht man solch ein Problem mit einer inline-Funktion, die typbehaftet ist:

```
inline int MAX( int a, int b ) {
    return a > b ? a : b; }
```

Damit können solche Probleme nicht auftreten. Allerdings leistet diese inline-Variante nicht so viel, wie das ursprüngliche MAX-Makro. Dieses konnte man mit beliebigen Datentypen benutzen, während die inline-Funktion nur für Integer nutzbar ist. Abhilfe

QUICK INDEX

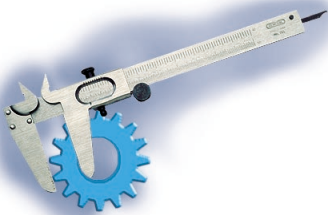
- #define vermeiden
- Speicher richtig allozieren
- Copy – Konstruktor und Zuweisung
- Basisklassen brauchen virtuelle Destruktoren
- Operator "=" ist nicht void
- Funktioniert "=" bei Selbstzuweisung
- Parameter "By reference" statt "By value"
- Überladen bei Zeigern und Integern
- Mehrfaches #include vermeiden



schafft ein C++-Template. Das folgende Template kann an allen Stellen benutzt werden, an denen auch das MAX()-Makro zum Einsatz kommen könnte – allerdings typensicher und ohne Nebeneffekte.

```
template <class T>
inline T& MAX( T&a, T&b) { return
    a>b ? a : b; }
```

Ordnung halten



■ Speicher richtig allozieren und freigeben

C++ kommt mit einem eigenen Konstruktor zum Anfordern und Freigeben von Speicher: `new` und `delete`. Diese beiden Operatoren sollte man auf jeden Fall verwenden – und nicht etwa die `malloc()` / `free()` Funktionsaufrufe aus der C-Run-time-Bibliothek. Diese stehen zwar prinzipiell zur Verfügung, es ist aber nicht geschickt, sie zu verwenden. Der Grund dafür ist einfach: `malloc()` und `free()` sind C-Funktionen, und als solche wissen sie nichts über Konstruktoren und Destruktoren. Wird also für ein Objekt per `malloc()`-Speicher angefordert, wird die Instanz des Objekts nicht richtig initialisiert, denn der Konstruktor des Objekts wird nicht aufgerufen: Malloc belegt einfach nur die angeforderte Größe an Speicher auf dem Heap. Das Gleiche gilt für `free()`. Wird ein Speicherblock per `free()` freigegeben, passiert genau dies und sonst nichts. Der Destruktor des Objekts wird nicht aufgerufen und alle dort durchgeführten Aufräumarbeiten bleiben unerledigt. Das schafft nicht nur unschöne Speicherlöcher, sondern auch sonst alle möglichen Resource-Lecks und andere unerwartete Seiteneffekte.

AUF WIEDERSEHEN

```
class Base {
    Base() {}
    ~Base() { printf("Und Tschuess!"); }
};

class Derived : public Base{
    Derived() {}
    ~Derived() { printf("Tschuess aus Derived!"); }
};
```

In diesem Zusammenhang ein Hinweis: Enthält eine Klasse Zeiger als Member, so müssen diese im Destruktor der Klasse mit `delete` zerstört werden. Anders als vielfach angenommen, werden nicht alle Destruktoren aller Member automatisch aufgerufen, sondern nur die Member, die keine Zeiger sind. Das ist sinnvoll, woher soll der C++-Compiler schließlich wissen, ob der Member-Pointer auf eine Kopie eines Objekts zeigt oder mehrere Pointer mehrerer Klassen-Instanzen auf das gleiche Objekt zeigen?

Für das Zerstören von Objekten, die in einer Klassen-Instanz per Member-Pointer erreichbar sind, ist die Klasse selbst zuständig.

Hier gilt es die normalen Regeln zu beachten. Nicht jeder Zeiger zeigt auf ein Objekt, das unbedingt zerstört werden muss!

Klassen vollständig ausstatten



■ Copy-Konstruktor und Zuweisungs-Operator

Wann immer man eine Klasse schreibt, in der Speicher dynamisch alloziert wird, muss man die Klasse mit einem Copy-Konstruktor und einem Zuweisungs-Operator ausstatten. Warum, das klärt das folgende Beispiel, das zu der XString()-Klasse passt. Die XString-Klasse soll dabei eine "typische" String-Klasse sein (siehe Listing "IE String-Klasse").

Die Klasse hat einen Konstruktor und einen Destruktor, aber keinen Zuweisungsoperator und keinen Copy-Konstruktor.

Im Folgenden sieht man recht schnell, welche Probleme dies bereitet:

```
XString a("C++ hat");
XString b("Fallstricke");
b = a;
```

IE STRING-KLASSE

```
Class XString {
private:
    char* pData;
public:
    XString( const char* p = 0);
    ~XString();
};
XString::XString( const char* p)
{
    if( p) {
        m_pData = new char[ strlen( p) + 1];
        strcpy( m_pData, p);
    }
    else {
        m_pData = new char[1];
        *m_pData = '\0';
    }
}
XString::~XString() { delete[] m_pData; }
```

Nachdem die beiden Konstruktoren durchgelaufen sind, hat sowohl die Instanz "a" als auch die Instanz "b" der Klasse Speicher alloziert – jeweils für den darin gespeicherten Text. In der dritten Zeile wird eine Zuweisung durchgeführt. Nachdem die Klasse aber keinen Zuweisungs-Operator definiert hat, wird der "normale" Weg von C++ verwendet, die Zuweisung kopiert die Member bitweise. Das hat unschöne Auswirkungen, denn nach der Zuweisung wird der ursprünglich von "b" allozierte Speicher zum Speicherleck. Es zeigt kein Zeiger mehr darauf, obwohl der Speicher nie freigegeben wurde. Außerdem zeigt `m_pData` in "b" auf den gleichen Speicherbereich wie `m_pData` in "a". Mit Sicherheit ist es nicht das, was der Programmierer beabsichtigte.

Ein ähnliches Problem tritt auf, wenn der Zuweisungs-Operator ins Spiel kommt. Angenommen man verwendet bei der gleichen Klasse den folgenden Ausdruck:

```
XString a("Hello World");
XString b = a;
```

tritt hier das gleiche Problem auf. Da die Klasse keinen Zuweisungs-Operator definiert hat, behandelt der Compiler die Zuweisung genauso wie zuvor den Kopiervorgang. Die Member werden bitweise kopiert und erneut zeigt `m_pData` von "b" auf die Daten aus "a". Nun mag das vielleicht in einer Implementierung Sinn machen, in diesem Fall ist aber zu beachten, dass der Destruktor der XString()-Klasse den Operator `delete` verwendet, um `m_pData` freizugeben. Nach der Zuweisung wird `delete` zweimal mit den gleichen Daten aufgerufen. Einmal im Destruktor von "a" und einmal in "b". Spätestens hier tritt also ein unerwarteter Nebeneffekt auf.



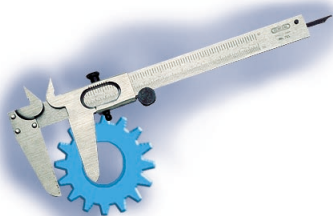
SELBST?

```
void XString::operator=( const char* p)
{
    m_pData = new char[ strlen( p) + 1];
    // etc. pp -
}
```

SELBST!

```
XString
XString::operator=( const char*
    p)
{
    // hier kommt der echte Code
    return *this;
}
```

Korrekt verabschieden



■ Basisklassen brauchen virtuelle Destruktoren

Angenommen man hat eine Basisklasse und eine abgeleitete Klasse und beide sollen sich in Ihren Destruktoren mit einer kurzen Meldung verabschieden (s. Listing "Auf Wiedersehen").

Nun sollen diese Klassen verwendet werden – und nachdem C++ praktischerweise Zeiger zu Instanzen von abgeleiteten Klassen in Zeigern auf Basisklassen verwalten kann, ist folgender Code völlig legal – und auch in vielen Fällen sinnvoll:

```
Base* pBase = new Derived;
// hier passiert etwas
// - was ist egal
delete pBase;
```

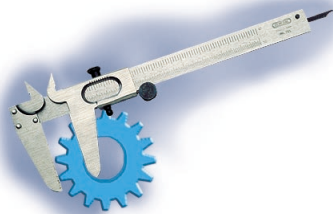
Wenn man die Ausdrücke in ein Beispielprogramm einbettet, übersetzt und ausführt, erhält man ein unbefriedigendes Resultat. Am Bildschirm wird nur der Text "Und Tschuess" angezeigt. Das erwartete "Tschuess aus Derived" erscheint einfach nicht (Listing auf S.87).

Der Grund dafür ist einfach. An der Stelle, an der per "delete" das Objekt zerstört werden soll, auf das pBase zeigt, muss der Compiler entscheiden, auf was für ein Objekt der Zeiger tatsächlich zeigt. Im Gegensatz zum Programmierer weiß der Compiler leider nicht, dass der pBase-Zeiger tatsächlich auf ein De-

rived-Objekt zeigt. Nur der Destruktor von der "Base"-Klasse wird aufgerufen – und das ist nicht das gewünschte Verhalten. Die Lösung ist einfach: Man muss den Destruktor der "Base"-Klasse

"virtual" machen – dann geht's so, wie man sich das vorstellen würde. Doch Vorsicht: Man ist nun leicht geneigt die Destruktoren aller Klasse grundsätzlich als "virtual" zu deklarieren, doch das sollte man besser bleiben lassen. Die Gründe dafür sind vielschichtig. Eine Auflistung würde den Rahmen dieses Artikels sprengen. Merken Sie sich daher die Faustregel, dass die Destruktoren von Basisklassen "virtual" sein müssen und die von anderen Klassen es nicht sein sollten.

Selbstbewusstsein ist wichtig



■ Operator"=" ist nicht void !

Es ist ein weit verbreiteter Irrtum, dass der Zuweisungs-Operator (Operator=") "void" ist. Der Grund für diesen Irrtum ist leicht gefunden, denn was man mit dem Operator= erreichen möchte, ist die Zulässigkeit von folgendem Ausdruck:

```
XString a;
a = "Hallo C++";
```

Auf den ersten Blick sieht es eindeutig so aus, als wäre der Operator= auch völlig ohne Rückgabewert prima zu implementieren (siehe Listing "Selbst?").

Hat man seine Klassen – und zwar zusammen mit dem Operator= für die einzelnen davon – implementiert, wird man über lang oder kurz folgenden Ausdruck hinschreiben wollen

– denn für die eingebauten Klassen wie "int" ist es legal:

```
XString a,b,c;
a = b = c = "Hello C++";
```

Das geht jedoch schief. Wie man in diesem Beispiel sehen kann, muss das Resultat einer Zuweisung eine Referenz auf den Wert der Zuweisung sein – ansonsten ist die verkettete Zuweisung nicht möglich. Lösung: Der Operator= muss einen Return-Wert haben, und dieser Return-Wert ist eine Referenz auf *this (s. Listing "Selbst!").

Begehen Sie keinen Selbstmord



■ Achtung: Funktioniert bei Selbstzuweisungen

Um diesen Fehler zu vermeiden, muss man zunächst wissen, was eine Selbstzuweisung ist. Das ist zum Glück einfach darzustellen – eine Selbstzuweisung sieht im einfachsten Fall wie folgt aus;

```
XString a;
a = a;
```

Zugegebenermaßen ist dieses Beispiel nicht sonderlich einleuchtend – allerdings ist es ein völlig legitimer Code, und daher sollte er auch funktionieren. Mit ein wenig Aufwand ist es einfach, ein überzeugenderes Beispiel zu finden:

```
a = b;
```

Zugegeben, der Aufwand, der betrieben werden muss, liegt beim Leser – denn

SELBSTAUSLÖSCHUNG

```
// .... code ..
delete [] m_pData;
m_pData = new char[ strlen( rCopy.m_pData) + 1];
strcpy( m_pData, rCopy.m_pData);
// ... mehr code....
```

SELBSTERHALTUNG

```
// zu Beginn des Operator=
XString& XString::operator=( const XString& rOther)
if( this == &rOther)
    return *this;
```

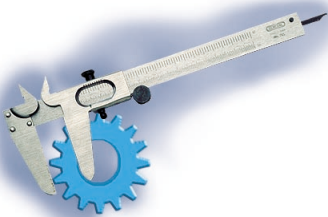


dieser muss sich nur vorstellen, dass a eine ganz normale Variable ist, während b ein Alias für "a" darstellt: zum Beispiel eine Referenz. Schon ist das Ganze durchaus plausibel und wird mit Sicherheit in der Praxis vorkommen – doch wo ist hier der Fallstrick?

Der Zuweisungs-Operator muss sich darum kümmern, dass vor der Zuweisung der neuen Daten die vorhandenen Ressourcen korrekt freigegeben werden. Er muss also etwas in der folgenden Art tun (s. Listing: "Selbstausslöschung"). Handelt es sich um eine Selbstzuweisung, so tritt an dieser Stelle sofort das Problem ans Licht. Die erste ausformulierte Zeile löscht den Daten-Buffer. Nachdem es sich aber um eine Selbstzuweisung handelt, ist dies der gleiche Puffer, aus dem in den nächsten Zeilen die Daten kopiert werden sollen. Das Objekt zerstört sich zunächst selbst und kopiert dann die defekten Daten auch noch weiter.

Man kann verschiedene Dinge tun, um solche Probleme zu vermeiden. So ist es zum Beispiel möglich, Zeilen auf gleiche Inhalte oder gleiche Zeiger zu überprüfen, bevor die Zuweisung wirklich durchgeführt wird: Welchen Weg man beschreitet ist dabei von den Details der Implementierung abhängig. Eine der Möglichkeiten sieht folgendermaßen aus (s. Listing "Selbsterhaltung").

Uraltes Problem immer aktuell



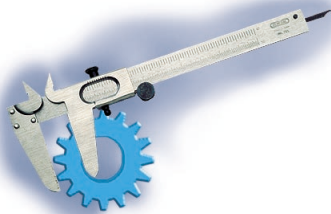
■ Parameter – "By reference" statt "By value"

Ähnlich wie in C ist die "Default"-Methode der Parameter-Übergabe in C++ "by value". Das bedeutet aber, dass bei jedem Funktionsaufruf von allen Parametern Kopien angelegt werden müssen. Das ist zwar bei eingebauten Datentypen wie einem Integer nicht weiter schlimm, bei komplexen Klassen hingegen schon. Angenommen eine Klasse verwendet dynamischen Speicher, dann hat der Copy-Konstruktor mit Sicherheit die Aufgabe, neuen Speicher zu allozieren und "alten" freizugeben. So

kann der Aufruf einer Funktion im schlimmsten Fall zum Pagen auf die Festplatte führen. Eine einfache Regel für C++ ist daher die: Wenn ein Parameter übergeben wird und wenn es sich bei dem Parameter um eine Instanz einer Klasse handelt, verwendet man entweder eine konstante Referenz – wenn das Objekt nicht verändert werden soll – oder einen Zeiger, wenn es doch verändert werden soll.

```
// Konstante Referenz
void FooBar( const & rObjekt );
```

Weniger ist mehr



■ Überladen – besser nicht mit Zeigern und Integer

Normalerweise meckert der C++ Compiler ja immer mehr Dinge als "ambiguous" an, als man sich wünschen würde – dummerweise gibt es aber außerdem noch einen Spezialfall, bei dem das genaue Gegenteil der Fall ist: Auf den ersten Blick hält der Programmierer die folgenden beiden Funktionen für eine unklare Art und Weise des Überladens, der Compiler hingegen nicht:

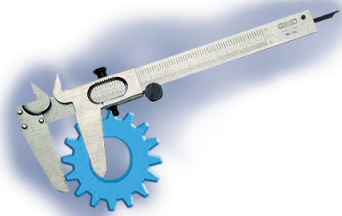
```
void fooBar( int v );
void fooBar( char* pv );
```

Die Frage, die sich bei diesen beiden Funktionen stellt, ist die: Wenn man im Quellcode den folgenden Aufruf verwendet:

```
fooBar ( 0 );
```

– welche der beiden Funktionen wird dann aufgerufen? Die Antwort lautet: *fooBar(int)*. Der Grund dafür ist einfach – auch wenn man auf den ersten Blick glaubt, das *fooBar(0)* nicht ohne weiteres aufgelöst werden kann, so ist bei näherem Hinsehen die Auflösung doch immer eindeutig. "0" ist eine Integer-Konstante – also ein "int" – und deshalb wird *fooBar(int)* auch aufgerufen – das "0" auch als Null-Zeiger verwendet werden kann (und wird) hat darauf keinen Einfluss. Eine einfache Lösung für dieses Problem gibt es nicht.

Einfach bleiben



■ Mehrfaches #includieren vermeiden

Ein typisches Problem bei C/C++ wird durch den Präprozessor hervorgerufen: das doppelte Inkludieren von Header-Dateien. Mal angenommen es existieren die beiden Header-Dateien „a.h“ und „b.h“ – sowie die Implementierungs-Datei „c.cpp“. a.h benötigt nun einige Definitionen aus b.h – und also enthält a.h am Anfang der Datei das Statement

```
#include "b.h"
```

Beim Implementieren von c.cpp hat der Programmierer diesen Umstand natürlich vergessen – und schreibt in c.cpp:

```
#include "a.h"
#include "b.h"
```

Das Problemist: b.h wird nun zweimal inkludiert – einmal indirekt über a.h, das andere Mal direkt durch das betreffende #include-Statement.

Daraus können eine ganze Menge Probleme entstehen – nicht zuletzt doppelt definierte Konstante und Ähnliches: Diese Art von Problem wird allerdings vom Compiler erkannt und bemängelt, sodass sich derartige Probleme meist mit ein wenig Nachdenken lösen lassen.

Dieses Problem – und auch das Problem mit den "doppelten" Konstanten – lässt sich aber einfach und dauerhaft lösen, wenn man sich an ein einfaches Schema hält: Jede Header-Datei definiert ein eindeutiges Symbol, und nur wenn dieses noch nicht definiert ist, wird der Inhalt der Datei tatsächlich an der Präprozessor – und damit später an den Compiler – weitergereicht. Klingt kompliziert – ist aber einfach und sieht in der Praxis wie folgt aus:

```
#ifndef _INC_NAME_DER_DATE
#define _INC_NAME_DER_DATE1

// hier steht dann der normale
// Inhalt

#endif
```

Auf diese Weise ist man immer auf der sicheren Seite. 