Effektiv C++ programmieren

22 Tips für besseres C++
basierend auf Ideen von Scott Meyers

Abstract

Auf Grundlage des Buches *Effective C++ - Second Edition* werden verschiedene Techniken zur korrekten Programmierung in C++ untersucht. Dabei geht der Autor insbesondere auf Unterschiede zu C ein, bespricht Aspekte der Konstruktion, Zuweisung und Destruktion von Klassen und schließt mit dem Entwurf und der Deklaration von Klassen ab.
Es wird vorausgesetzt, dass der Leser grundlegende Kenntnisse in C++ bzw. einer eng verwandten Sprache wie C oder Java besitzt. Ebenso sollten die Prinzipien der objektorientierten Programmierung geläufig sein.

Gliederung

Einführung

Die Festlegung einer einheitlichen Terminologie ist für das Verständnis unabdingbar. Eine *Deklaration* teilt dem Compiler Namen und Typ eines Objektes, einer Funktion, einer Klasse oder eines Templates mit:

```
extern string strHostname;              // Objektdeklaration
string GetHostname(int nLocalIP);       // Funktionsdeklaration
class CServer;                          // Klassendeklaration
template<class T> class CStack;         // Template-Deklaration
```

Die Deklaration gibt aber keinerlei Auskunft über den genauen Aufbau oder die Struktur. Dies ist Aufgabe der *Definition*. Eine Objektdefinition erlaubt dem Compiler, Speicher bereitzustellen, hingegen eine Funktionsdefinition (auch eine Funktions-Templatedefinition) enthält den kompletten Funktionsrumpf, d.h. den auszuführenden Code. Die Definition einer Klasse oder eines Klassen-Templates zählt die Elemente derselben auf:

```
string strHostname;                     // Objektdefinition

string GetHostname(int nLocalIP)        // Funktionsdefinition
{
    return „www.hpi.uni-potsdam.de“;
}

class CServer                           // Klassendefinition
{
    int m_nPort;

public:
    CServer();
    CServer(const CServer& server);
    virtual ~CServer();
    string GetHostname(int nLocalIP) const;
    void Listen();
};

template<class T>                       // Template-Definition
class CStack
{
    vector m_Data;

public:
    CStack(int nMaxSize = 10);
    ~CStack();
    T Pop();
    Push(const T& element);
};
```

Jede Klasse verfügt über mindestens einen *Konstruktor*, welcher bei der Initialisierung aufgerufen wird. Ein *Default-Konstruktor* ist derjenige, der entweder keine Parameter benötigt (erster Konstruktor von `CServer`) oder für jeden Parameter einen Default-Wert kennt (`CStack`). Für ein Array von Objekten ist ein Default-Konstruktor unabdingbar, aber auch viele Objekt-Erzeugungsmechanismen - wie z.B. Serialisierungsverfahren - sind auf den Default-Konstruktor angewiesen. Der *Copy-Konstruktor* dient dazu, ein Objekt mit einem anderen vom gleichen Typ zu initialisieren (zweiter Konstruktor von `CServer`). Er wird u.a.

auch aufgerufen, wenn die Übergabe eines Objekt als Wertparameter und nicht per Referenz erfolgt sowie bei der Rückgabe eines Ergebnisobjektes. Manchmal muss der Compiler zur Auflösung komplexer Ausdrücke temporäre Objekte erzeugen. Diese generiert er wiederum über den Copy-Konstruktor.

Bei der Einführung des Konstruktors fiel der Begriff *Initialisierung*. Ein Objekt wird initialisiert, wenn es zum allerersten Mal einen Wert bekommt, für Objekte mit Konstruktor(en) wird stets ein Konstruktoraufruf durchgeführt. Eine *Zuweisung* hingegen weist einem bereits existierendem Objekt einen neuen Wert zu. Der technische Unterschied zwischen den folgenden Zeilen Code besteht darin, dass die ersten beiden den Copy-Konstruktor verwendet, während die letzte den `operator=` benutzt:

```
CPoint Origin(Center);
CPoint Origin2 = Center;
Origin2 = Center;
```

Die Differenzierung ist erforderlich, da ein bereits existierendes Objekt wahrscheinlich Ressourcen alloziiert hat. Diese muss die Zuweisung wieder freigeben, um Lecks zu verhindern. Auf der anderen Seite sollte eine Initialisierung die Gültigkeit ihrer Argumente stets überprüfen, während eine Zuweisung von der Gültigkeit ausgehen kann (da das Argument selbst bereits initialisiert sein muss).

Der Umstieg von C nach C++

Tip 1: Vermeidung von Problemen des Präprozessors

Die Sprache C besitzt nur wenige Möglichkeiten, um häufig benutzte Codefragmente (Makros) wiederzuverwenden. Dieses Manko wird durch die Verwendung des Präprozessors umgangen, der lediglich eine einfache Textersetzung durchführt. Der bearbeitete Sourcecode wird danach an den Compiler weitergereicht, welcher aber keinerlei Kenntnisse mehr über die ursprünglichen Makros besitzt. Bei Fehlern kann er sich daher nicht auf den vom Programmierer geschriebenen Code beziehen sondern nur auf die vom Präprozessor modifizierten Zeilen.

In C++ ist es erlaubt, Konstanten explizit zu definieren. Neben dem Schlüsselwort `const` wird noch ein Datentyp verlangt, so dass der Compiler sofort eine Typüberprüfung durchführen kann. Bei klassenspezifischen Konstanten muss eine Deklaration *und* eine Definition erfolgen:

```
class CAnswer                          // in der Headerdatei
{
    static const int ANSWER;
    ....
};

const int CAnswer::ANSWER = 42;        // in der Quelldatei
```

Makros dienen oftmals auch dazu, kurze generische Funktionen inline zu verwenden, d.h. den kompletten Code an Ort und Stelle einzufügen, statt die Funktion aufzurufen. Dabei können gefährliche Seiteneffekte auftreten, die darauf beruhen, dass Makros eben lediglich eine einfache Textersetzung darstellen und deshalb Code in den Parametern unbewusst mehrfach generiert wird. Das passiert besonders oft bei Inkrement-/Dekrementoperationen, wie `a++`. Folgendes Beispiel stellt ein teilweise fehlerhaft arbeitendes Makro einem stets korrekten Template gegenüber:

```
#define max(a,b) ((a) > (b) ? (a) : (b)) // C-Makro

template<class T>                        // Template in C++
inline const T& max_Template(const T& a, const T& b)
{
    return a > b ? a : b;
}

int a=b=c=d=5;
int c    = max(++a,  0);               // c=a=7, aber a=6 erwartet
int c2   = max(++b, 10);               // c2=10, b=6, korrekt
int cpp  = max_Template(++c,  0);      // cpp=c=6, korrekt
int cpp2 = max_Template(++d, 10);      // cpp2=10, d=6, korrekt
```

In C++ stehen Templates zur Verfügung, die der Compiler selbst auswertet und damit das gesamte Probleme der Textersetzung elegant umgeht. Im Gegensatz zu Makros verfügen Templates zusätzlich über die Eigenschaft, dass sie Typverträglichkeiten überprüfen können. Unter Verwendung des Schlüsselwortes `inline` treten beim Einsatz von Templates in der Regel keine Laufzeiteinbußen gegenüber herkömmlichen Makros auf.

Tip 2: Kommentare

Zusätzlich zu dem C-Kommentar, der mit `/*` beginnt und alle Zeichen (zeilenübergreifend) bis zum nächsten `*/` ignoriert, kann man in C++ mit `//` einen Kommentar definieren, der immer nur bis zum Zeilenende geht und durch nichts vorher unterbrochen werden kann, auch nicht durch weitere Kommentarsymbole.

```
// double dCash = 500.0; // auch ein zweiter Kommentar ist erlaubt
/* double dCash = 500.0; /* verschachtelte C-Kommentare sind verboten */ */
```

Tip 3: Neue I/O-Routinen

Bei der Ein- und Ausgabe von Daten ist es in C++ guter Stil, auf `printf`, `scanf` und ihre Derivate zu verzichten. Statt dessen benutzt man die Operatoren << und >>, welche für alle Basistypen in der Headerdatei iostream definiert sind. Sie erlauben den Verzicht auf fehleranfällige Formatierungscodes wie `%d`, `%n` etc.
Ein kleiner Hinweis: in iostream wird durchgängig der Namensraum `std` verwendet, er ist ggf. zu öffnen bzw. als Präfix zu verwenden (siehe auch Tip 22).

Tip 4: Geänderte Speicherverwaltung

Die aus C bekannten Funktionen `malloc` und `free` (sowie deren Abkömmlinge) sind nicht in der Lage mit Konstruktoren und Destruktoren umzugehen, weshalb sie für objektorientierte Programme ungeeignet sind. Die neu in C++ eingeführten Operatoren `new` und `delete` können dagegen sogar für jede Klasse einzeln überladen werden. Es sollte auf keinen Fall ein mit `malloc` angelegter Speicherbereich mit `delete` freigegeben werden. Ebenso ist das Ergebnis undefiniert, wenn auf `new` ein `free` folgt.

Tip 5: Sichere Typkonvertierungen

In C++ erlauben neue Casts, dass eine statische oder dynamische Überprüfung der Typverträglichkeit vorgenommen werden kann. Es ist zusätzlich noch die aus C bekannte Cast-Vorgehensweise erlaubt, sie sollte aber nicht mehr verwendet werden.

Am häufigsten kommt sicherlich ein `dynamic_cast` in Frage, er dient zur sicheren Durchführung eines Downcasts, d.h. von einer Basisklasse auf eine davon abgeleitete Klasse. Die Basisklasse selbst muss polymorph sein, also mindestens eine virtuelle Methode besitzen. Schlägt der Cast fehl, so liefert er `NULL` (Zeiger-Cast) bzw. eine `bad_cast`-Exception (Cast einer Referenz) zurück. Da die Überprüfung erst zur Laufzeit durchgeführt werden kann, ist mit `dynamic_cast` ein gewisser Laufzeit-Overhead in Form von Runtime Type Information (RTTI) verbunden, bei Visual C++ ist gar eine explizite Aktivierung notwendig.

Eine Typkonvertierung zum Zeitpunkt der Kompilierung wird meist mit dem Schlüsselwort `static_cast` vorgenommen. Es ist jedoch zu beachten, dass dem Compiler nicht allzu viele Informationen über das umzuwandelnde Objekt zur Verfügung stehen und er in der Regel nur

auf Grundlage der statischen Klassenhierarchien arbeiten kann. `static_cast` sind wie `dynamic_casts` portabel.

Um ein als `const` gekennzeichnetes Objekt verändern zu dürfen, existiert der `const_cast`. Es wird jedoch empfohlen, diesen Cast nur in äußerst seltenen Fällen anzuwenden, da oft schon ein als `mutable` deklariertes Attribut das Problem sauber lösen kann. Tip 16 geht genauer darauf ein.

Der unsicherste und in der Regel nicht portable Konvertierungsmechanismus ist der `reinterpret_cast`. Er ist dann erfolgreich, wenn die Bitbreiten von Ausgangs- und Zieltyp übereinstimmen. Die Auswertung erfolgt zum Übersetzungszeitpunkt, Klassenbeziehungen werden ignoriert.
Die Schreibweise der neuen C++-Casts folgt dem Schema:

```
Ziel = static_cast<Zieltyp>(Quelle);
```

Als Beispiel für einen Downcast:

```
CBase* derived = new Derived;
CDerived* derived2 = dynamic_cast<CDerived*>(derived);
```

Konstruktoren, Destruktoren und Zuweisungsoperatoren

Tip 6: Initialisierung vs. Zuweisung im Konstruktor

Bei Konstruktoren ist es erlaubt und in gewissen Fällen sogar erforderlich, dass Attribute initialisiert werden, bevor auch nur die erste Zeile Code des Konstruktors ausgeführt wird. Die Syntax dafür lautet:

```
CClass::CClass() : Attribut1(Wert1), Attribut2(Wert2)
{
    ...
}
```

Der Wert kann sich ggf. aus einer Formel ergeben, wenn dabei Attribute der Klasse verwendet werden, dann ist die Reihenfolge der Initialisierung zu beachten (siehe Tip 7).

Falls Attribute eine Referenz oder als `const` deklariert sind, dann *muss* man eine Initialisierung durchführen. In vielen anderen Fällen, wo auch eine Zuweisung erlaubt ist, greift man trotzdem auf eine Initialisierung zurück: sie ruft lediglich den entsprechenden Copy-Konstruktor auf. Eine Zuweisung geht dagegen mit einem Aufruf vom Defaultkonstruktor und vom `operator=` einher, was in der Regel kostspieliger ist.

Tip 7: Reihenfolge der Initialisierung im Konstruktor

Egal in welcher Abfolge die Datenelemente zur Initialisierung bei der Implementation eines Konstruktor auftauchen, sie werden stets in der Reihenfolge initialisiert, in der sie *deklariert* wurden. Die Kenntnis über diese Vorgehensweise ist wichtig, wenn in der Initialisierung einzelne Attribute von anderen abhängen. Zusätzlich erfolgt die Bearbeitung aller Datenelemente der Basisklasse *vor* denen der abgeleiteten Klasse. Es ist daher empfehlenswert, die exakte Deklarationsreihenfolge auch bei der Initialisierung des Konstruktors zu verwenden, um Verwirrung zu vermeiden.

Tip 8: Virtuelle Destruktoren

Ein Destruktor hat die Aufgabe, alle Ressourcen eines Objektes vor dessen Zerstörung freizugeben und etwaige Aufräumarbeiten zu erledigen. Geht durch Casts auf eine Basisklasse der genaue Datentyp verloren, so ist `delete` nicht mehr in der Lage, den korrekten Destruktor auszuwählen. Es kann dann passieren, dass nicht alle Ressourcen bereinigt werden oder gar völlig falsche Aktionen zur Ausführung kommen.

Die Lösung für dieses Problem in C++ ist die Deklaration eines *virtuellen* Destruktors. Anhand der virtuellen Methodentabelle ist der Compiler dann stets in der Lage, den korrekten Destruktor zu finden, selbst wenn er nicht mehr die genaue Klasse eines Objektes kennt. Genau genommen ist das Schlüsselwort `virtual` nur beim Destruktor der Basisklasse wichtig, der Compiler merkt sich dies und behandelt automatisch alle Destruktoren der abgeleiteten Klassen virtuell. Zum besseren Verständnis sollte man trotzdem `virtual` überall dort schreiben, wo es auch implizit durch Vererbung folgt.

Leider ist für den Zugriff auf die virtuelle Methodentabelle pro Objekt jeweils ein Zeiger notwendig, auf heutigen Rechnern stellt dies einen zusätzlichen Speicherverbrauch von 4 Byte dar. Gerade für häufig benutzte Klassen mit wenigen Attributen ist dies ein wichtiges Kriterium. Ebenso sorgen die nur indirekt durchgeführten Methodenaufrufe für einen leicht erhöhten Laufzeitbedarf.
Es ist ratsam, alle nicht-leichtgewichtigen Klassen, die als Basisklassen dienen sollen, mit einem virtuellen Destruktor auszustatten. Dies gilt insbesondere auch für abstrakte Klassen.

Tip 9: Deklaration eines Copy-Konstruktors und Zuweisungsoperators für Klassen mit dynamisch alloziiertem Speicher

Für jede Klasse, die einen Copy-Konstruktor oder Zuweisungsoperator benutzt, existieren diese, auch wenn sie nicht explizit angelegt werden. Der C++ Compiler generiert selbstständig Code, der eine bitweise Kopie erzeugt. In vielen Fällen ist dies erwünscht und korrekt. Man spart Tipparbeit und umgeht das Problem, dass man später zu einer Klasse hinzugefügte Attribute schlichtweg vergisst zu kopieren.

Wenn nur ein einziges Attribut ein Zeiger ist, dann stellt eine „bitweise Kopie“ ein Problem dar: es wird die Adresse des Objektes, auf das man zeigt, kopiert, nicht aber das Objekt selber. Aus diesem Grunde muss jede Klasse, die über ihre Attribute dynamisch Speicher alloziiert, über einen Copy-Konstruktor und einen Zuweisungsoperator verfügen.

Das „Bitweise Kopie“-Problem kann insbesondere zuschlagen, wenn Parameter als Wert übergeben werden (siehe Einführung), weil dann oft ein impliziter Copy-Konstruktoraufruf geschieht. Tip 21 beschreibt eine Technik, wie man den Copy-Konstruktor und den Zuweisungsoperator verbieten kann.

Tip 10: Der Copy-Konstruktor und der Zuweisungsoperator im Zusammenspiel mit Klassenhierarchien

Gutes Klassendesign fordert, dass Attribute als `private` deklariert werden. Ist man gezwungen, wie im vorherigen Tip, einen eigenen Zuweisungsoperator/Copy-Konstruktor zu schreiben, so hat man dann aber keinen Zugriff auf die Attribute der Basisklasse mehr. Es ist daher unabdingbar, in `operator=` auf jeden Fall den Zuweisungsoperator der Basisklasse aufzurufen, in der Initialisierungsliste des Copy-Konstruktors muss ebenfalls die Basisklasse angegeben werden.

```
// Copy-Konstruktor
CDerived(int a, int b) : CBase(a)
{
    // b kopieren
    ...
}
```

```
// Zuweisungsoperator
CDerived& CDerived::operator=(const CDerived& obj)
{
    CBase::operator=(obj);
    // Attribute von CDerived kopieren
    ...
}
```

Tip 11: Der Rückgabewert des Zuweisungsoperators

Für die eingebauten Datentypen sind in C++ viele Varianten einer Zuweisung möglich:

```
int x, y;
x = y = 0;                          // entspricht x = (y = 0)
(x = y) = 1;                        // entspricht x = y; x = 1;
```

Die einzige Möglichkeit, diese Verkettungen für eigene Klassen ebenfalls zu erlauben, besteht darin, dass der Rückgabewert des Zuweisungsoperators das in der Anweisung *links* stehende Objekt ist. Dafür bietet sich eine Referenz auf die eigene Klasse an. Damit die dritte Zeile auch funktioniert, darf sie nicht `const` sein.

Die letzte Zeile eines Zuweisungsoperators lautet immer `return *this;`. Im vorangegangenen Tip zum Thema Copy-Konstruktoren existiert im Beispielcode ein dementsprechend deklarierter Operator.

Hinweis: Meine persönliche Meinung ist, dass der Rückgabewert doch `const` sein sollte, da das Beispiel `(x=y)=1;` doch sehr selten auftritt und schlechten Stil darstellt.

Tip 12: Prüfung auf Zuweisung an sich selbst

In C++ ist es vollkommen legal, `x=x;` zu schreiben. Neben diesem sehr offensichtlichen Gebilde gibt es aber auch viele subtilere, die über Indirektionen auf eine Zuweisung an sich selbst hinauslaufen, sie werden als Aliasing-Problem bezeichnet und können unter Umständen gar nicht verhindert werden.

Das Problem besteht darin, dass eine Zuweisung alle bisher von der linken Seite belegten Ressourcen freigeben will und danach die der rechten Seite kopiert. Genau das scheitert bei einer Zuweisung an sich selbst, die zu kopierenden Ressourcen sind gar nicht mehr vorhanden (sie wurden schließlich freigegeben).

Ein einfacher Weg zur Bestimmung von Objektidentität ist der Vergleich der Speicheradressen. Die erste Codezeile im Zuweisungsoperator sollte also stets lauten:

```
CClass& CClass::operator=(const Class& obj)
{
    if (this == &obj)
        return *this;
    ...
}
```

Es ist daran zu denken, dass quasi *jede* Methode, die als Parameter eine Referenz oder einen Zeiger auf ihre eigene Klasse akzeptiert, auch das eigene Objekt übergeben bekommen könnte. Dann gelten die gleichen Aussagen wie für den Zuweisungsoperator.

Entwurf und Deklaration von Klassen und Funktionen

Tip 13: Vollständige aber minimale Schnittstellen

Eine Klasse sollte nur wirklich die Methoden über ihre Schnittstelle bereitstellen, die zur vollständigen Erfüllung ihrer Aufgabe notwendig sind. Widersetzt man sich dieser Faustregel, dann erhöht sich neben der Dauer eines Kompiliervorganges (aufgrund langer Headerdateien) vor allem die Einarbeitungszeit der Programmierer, da die Übersicht verloren geht. Ebenso ist es schwer, bei Erweiterungen die Klasse vernünftig zu warten, da oft unerwartete Seiteneffekte auftreten.

Auf der anderen Seite sollte eine Klasse minimal über einen Konstruktor und einen Destruktor verfügen. In den meisten Fällen ist auch ein Copy-Konstruktor und ein Zuweisungsoperator notwendig, fast alle Klassenbibliotheken empfehlen die Überprüfung der Klasseninvariante und einen Serialisierungsmechanismus.

Tip 14: Vermeidung von Attributen in der öffentlichen Schnittstelle

Mit jeder Veröffentlichung eines Attributes in der öffentlichen Schnittstelle einer Klasse verliert man die Kontrolle über Schreib- und Lesezugriffe darauf. Diese können zu unberechtigten Aktionen oder Inkonsistenzen führen. Weiterhin ist es unmöglich, die Zuweisung später durch eine Berechnung plus Zuweisung ersetzen zu können (z.B. eine Protokollierung aller Zugriffe).

Das objektorientierte Konzept der funktionalen Abstraktion fordert, dass jedes Attribut, das unbedingt öffentlich zugreifbar sein soll, durch entsprechende Lese- und Schreibmethoden (`Get/Set` oder `Read/Write`) zu ersetzen ist. Mit `inline` kann der entstehende Overhead im kompilierten Programm meist eliminiert werden.

Tip 15: Element-Funktionen, globale Funktionen und `friend`*-Funktionen*

Viele Methodenaufrufe folgen dem Schema:

```
result = obj1.Methodenname(obj2);
noon   = morning.wait(6);
noon   = morning + 6;                  // entspricht morning.operator+(6)
```

Um diese Zeile übersetzen zu können, schaut der Compiler in der Klasse von `obj1` nach, ob `Methodenname` mit passender Signatur existiert. Für die letzten beiden Zeilen müssen `wait` und `operator+` für die Klasse von `morning` definiert sein. Idealerweise geschieht dies als Elementfunktion, beide Methoden sind *Bestandteil* der Klasse.

Leider ist entgegen den Gesetzen der Mathematik die Umkehrung nicht möglich, d.h. `6+morning` kann *nicht* ausgewertet werden, da 6 keiner bekannten Klasse zugehört. Ein Ausweg besteht darin, `operator+` für diese Datentypen als globale Funktion zu definieren:

```
const CTime operator+(int start, const CTime& offset)
{
    ...
}
```

Über implizite Umwandlungen von `int` nach `CTime` mit Hilfe des Konstruktors der Klasse `CTime` lässt sich der Datentyp des ersten Parameters auf `CTime` verallgemeinern, der Compiler kann dann selber die `6` implizit konvertieren. Im Tip 20 folgt aber ein Hinweis, warum zu viele Typumwandlungen gefährlich sein können.

Unter gewissen Umständen kann es notwendig sein, dass eine globale Funktion Zugriff auf als `private` deklarierte Attribute benötigt. Diesen kann sie erhalten, wenn man über `friend` eine Ausnahmegenehmigung erteilt:

```
class CTime
{
    ...
    friend operator+(int, const& CTime);  // ist eine globale Funktion
};
```

Zusammenfassend gibt es als Entscheidungshilfen einige Grundregeln:
- virtuelle Funktionen *müssen* Elementfunktionen sein
- `operator<<` und `operator>>` sind *immer* globale Funktionen, ggf. benötigen sie eine `friend`-Deklaration
- nur globale Funktionen erlauben eine implizite Typumwandlung des linken Arguments
- alle andere Funktionen sollten Elementfunktionen sein

Tip 16: Möglichst häufige Benutzung von `const`

Die Bedeutung von `const` besteht in der Unveränderbarkeit des attributierten Symbols. Innerhalb einer Klassendeklaration kann eine Methode `const` sein, wenn sie kein Attribut dieser Klasse verändert. Der Compiler generiert dadurch keinen anderen Code, er kann aber unbeabsichtigte Wertveränderungen entdecken. Mit `const` schützt sich der Programmierer vor sich selbst. Sollte ein Attribut auch durch `const`-Methoden manipuliert werden dürfen, so muss dieses Attribut als `mutable` deklariert werden.

Wichtig ist die Unterscheidung zwischen bitweiser und konzeptioneller Konstantheit: C++ kümmert sich lediglich um ersteres. Eine `const`-Methode kann ein Zeiger-Attribut nicht verändern. Jedoch ist es erlaubt, dass das, worauf dieser Zeiger verweist, schreibend bearbeitet werden darf.

Es ist zu beachten, dass bei einem mit `const` definierten Objekt der Datentyp ein anderer ist als ohne, d.h. `const CClass` ist etwas anderes als nur `CClass`. Bemerkbar macht sich dies bei der Klasse `CTime` aus dem vorangegangenem Tip. Nur weil der Rückgabewert als `const` vereinbart wurde, kann der Compiler Zeilen wie

```
(morning+noon)=evening;              // einem temporären Objekt wird
                                     // evening zugewiesen
```

verbieten. Hätte der Programmierer das `const` vergessen, würde der Compiler anstandslos diese Zeile übersetzen.

Bei Zeigern gibt es drei Fälle von `const`: (I) das, worauf gezeigt wird, darf nicht verändert werden, (II) der Zeiger selbst darf nicht verändert werden und (III) weder der Zeiger noch der Inhalt darf geändert werden.

```
#define ANSWER 42                       // C-Stil, nicht mehr benutzen !
const int ANSWER = 42;                  // C++, einfache Integer-Konstante
const char* strAnswer1 = „zweiundvierzig“;        // Zeiger, I
char* const strAnswer2 = „zweiundvierzig“;        // Zeiger, II
const char* const strAnswer3 = „zweiundvierzig“;  // Zeiger, III
```

Tip 17: Übergabe via Wert vs. Übergabe via Referenz

In einer Parameterliste sorgt eine konstante Referenz dafür, dass der Copy-Konstruktor nicht aufgerufen werden muss, was sich gerade bei großen Objekten in der Laufzeit stark positiv bemerkbar macht.

Zusätzlich wird das sogenannte Slicing-Problem unterbunden: es bezeichnet den Umstand, dass bei einer Konvertierung in eine Basisklasse alle diejenigen Eigenschaften verloren gehen, die die abgeleitete Klasse ausgezeichnet haben. Bei der Referenzübergabe jedoch bleibt der originale Datentyp erhalten, selbst wenn in der Parameter-Liste nur eine Basisklasse verlangt wird. Bei einer Wertübergabe wäre dies nicht der Fall gewesen.

Technisch geschieht die Übergabe einer Referenz mittels eines Zeigers, wodurch man eine Indirektion einführt. Für die meisten direkt in C++ eingebauten Datentypen, wie `int`, ist der Kopieraufwand aber extrem gering und in diesen Fällen sollte man auf eine Übergabe via Referenz verzichten.

Tip 18: Gefährliche Rückgabe einer Referenz

Es wurde bereits diskutiert, dass der Zuweisungsoperator eine Referenz als Rückgabewert besitzen sollte. Es ist dabei stets die Existenz des Objekts, auf das verwiesen wird, sichergestellt. Anders verhält es sich, wenn ein dynamisch oder lokal erzeugtes Objekt referenziert wird. Dessen Lebensdauer ist meist vollkommen unklar, ebenso ist oft nicht zweifelsfrei ermittelbar, wieviele Referenzen auf dieses Objekt existieren und wer für die Zerstörung zuständig ist. Aus diesem Grunde sollten neu erzeugte Objekte stets als Wert zurückgegeben werden.

```
// falsche Rückgabe einer Referenz
CNumber& CNumber::operator+(const CNumber& a, const CNumber& b)
{
    CNumber result(a+b);
    return result;                      // result wird zerstört, sobald das
                                        // Programm Sichtbarkeitsbereich von
                                        // operator+ verlässt
}
```

```
// erneut fehlerhaft, da Zerstörung von pResult ungewiss ist
CNumber& CNumber::operator+(const CNumber& a, const CNumber& b)
{
    CNumber *pResult = new CNumber(a+b);
    return result;
}

CNumber x,y,z;
CNumber sum = x+y+z;                    // y+z hinterlässt Speicherleck
```

Korrekt wäre die zuerst aufgeführte Addition wenn man den Rückgabewert *nicht* als Referenz deklariert.

Tip 19: Unterscheidung zwischen Überladung und Default-Parametern

Wenn eine Methode, die unter dem gleichen Namen in einer Klasse mehrfach auftaucht und sich nur in der Anzahl der Parameter unterscheidet, dann hängt die Wahl zwischen Überladung und Default-Parametern vom benutzten Algorithmus ab. Ist er überall identisch, so stellen Default-Parameter die optimale Wahl dar, ansonsten ist Überladung zu bevorzugen.

Tip 20: Quellen von Mehrdeutigkeiten

Im C++-Standard sind viele Typkonvertierungen implizit vorgegeben, sie können zusätzlich vom Programmierer durch `operator Zieltyp()` und Konstruktoren in einer Klasse erweitert werden. In einigen Fällen ist es unerwünscht, dass diese Konvertierungen durchgeführt werden, weil dadurch Mehrdeutigkeiten entstehen.
Folgender Code wird problemlos übersetzt:

```
double x = 'A';                         // implizit: char => double
cout << x;                              // Ausgabe: 65, ASCII-Code von 'A'
```

Ein als `explicit` deklarierter Konstruktor wird nur dann benutzt, wenn man ihn auch explizit aufruft. Der Compiler versucht nicht, ihn für implizite Typkonvertierungen zu verwenden:

```
class CTime
{
public:
    explicit CTime(double dHours) { ... } // expliziter Konstruktor
    SetTime(const CTime& time) { ... }
    ...
};

CTime obj ;
obj.SetTime(12.0);                      // Fehler, da 12.0 nicht vom Typ
                                        // CTime ist und Konstruktor nicht
                                        // implizit benutzt werden darf
obj.SetTime(CTime(12.0));               // korrekt
```

Mehrfachvererbung kann viele Mehrdeutigkeiten provozieren. Insbesondere die Deklaration von signaturgleichen Methoden in den Basisklassen stellt ein Problem dar und sollte möglichst vermieden werden.

Tip 21: Verbot automatisch generierter Methoden

In C++ enthält jede Klasse automatisch den Copy-Konstruktor und den Zuweisungsoperator, selbst wenn diese nicht explizit programmiert wurden. In einigen Fällen ist das unerwünscht. Ein Verbot erreicht man, indem beide als `private` deklariert, aber nicht definiert werden:

```
// Objekte dieser Klasse können nicht kopiert werden
class CSealed
{
public:
    // Default-Konstruktor
    CSealed() {}

private:
    // Copy-Konstruktor verbieten
    CSealed(const CSealed& obj);
    // Zuweisungsoperator verbieten
    operator=(const CSealed& obj);
};
```

Tip 22: Unterteilung des globalen Namensraums

In großen Projekten ist es oft unumgänglich, dass Namen mehrfach verwendet werden, z.B. wenn in jeder Bibliothek eine Konstante `VERSION` erzeugt wird. Da dies der Compiler bemängelt, kann man mit `namespace` einen untergeordneten Namensraum schaffen. Ähnlich wie bei Klassen ist mit dem Scope-Operator `::` zu arbeiten.

Wenn die stete Benutzung des Scope-Operators zu aufwändig ist, dann kann man mit `using Symbol` ein Symbol gezielt aus dem fremden Namensraum in den eigenen laden. `using namespace Name` erlaubt das gleiche gar für einen kompletten Namensraum. Beide Techniken sind nur benutzbar, wenn keine Namenskollisionen auftreten.

Die C++-Standardbibliothek ist im Namensraum std untergebracht:

```
std::cout << „Hallo “;                  // Namensraum explizit angeben

using std::cout;                        // nur cout aus std importieren
cout << „Welt !“;

using namespace std;                    // kompletten Namensraum einbinden
cout << endl;                           // auch end list damit verfügbar
```

Literatur

Scott Meyers: Effektiv C++ programmieren, 3.Auflage, Addison-Wesley Longman, Bonn, 1998
Hinweis: Die Nummerierung in diesem Buch unterscheidet sich von meiner, die grobe Reihenfolge der Tips ist aber ähnlich.