

3. Wichtige Begriffe und Sprachelemente

3.1 namespace und using

Problem:

Viele Programmierer → Viele Header-Dateien → Viele gleiche Namen

Abhilfe:

Jeder Programmierer vergibt ein eindeutiges Präfix für seine Namen. Statt dies von Hand zu tun, benutzt man namespace.

Statt:

```
static const double peterdVERSION = 2.0;
class peterMyClass
{
    ...
};
```

Besser:

```
namespace peter
{
    static const double dVERSION = 2.0;
    class MyClass
    {
        ...
    };
}
```

Nun muss der Nutzer natürlich auch dieses Präfix benutzen, um an seine Namen zu kommen:

a) **Explizite** Angabe des Präfixes der Namen:

```
double dVersion = peter::dVERSION;
peter::MyClass Obj;
```

b) **Implizite** Benutzung des Präfixes **für ausgewählte Namen (using)**:

```
using peter::dVERSION;
double dVersion = dVERSION;
```

c) **Implizite** Benutzung des Präfixes **für alle Namen (using)**:

```
using namespace peter;
double dVersion = dVERSION;
MyClass Obj;
```

Man sollte es jedoch **vorziehen**, eine **vertikale Teilung der Software mittels Client/Server-Modell** durchzuführen, was die Verwendung von gleichen Namen erlaubt.

3.2 Default-Konstruktor

Konstruktor ohne Argumente oder ausschließlich mit Default-Argumenten → Default-Initialisierung. Beispiel:

```
class MyClass
{
    public:
        MyClass() : m_nA(0),m_nB(1) {}
    private:
        int m_nA;
        int m_nB;
};
```

oder (hier ist der normale Konstruktor auch schon enthalten):

```
class MyClass
{
    public:
        MyClass(int nA = 0,int nB = 1)
            : m_nA(nA),m_nB(nB) {}
    private:
        int m_nA;
        int m_nB;
};
```

Anwendung: MyClass Obj;

3.3 Copy-Konstruktor

Konstruktor zur Initialisierung eines Objektes mit Hilfe der Werte eines anderen Objektes → Es wird ein Objekt der gleichen Klasse als Argument übergeben. Beispiel:

```
MyClass::MyClass(const MyClass& Obj)
{
    if(this == &Obj)
        return;
    ...
}
```

Anwendung: MyClass ObjA;
 MyClass ObjB(ObjA);

3.4 explicit-Konstruktor

Konstruktor, der **nicht** zur impliziten Typ-Konvertierung herangezogen werden kann

→ Schlüsselwort `explicit`.

Hintergrund:

Normalerweise kann der Compiler **statt einem Objekt als Argument einer Funktion** auch den **Parameter dessen Konstruktors** entgegennehmen, vorausgesetzt der Konstruktor hat **nur 1 Argument**. Findet er eine solche Stelle, dann fügt er dort den Code für die Konstruktion des Objektes mit diesem Argument ein (implizite Typumwandlung).

Möchte man diese implizite Typkonvertierung abschalten, dann ist der Konstruktor mit `explicit` zu deklarieren.

Beachte:

`explicit` darf in der Implementierung nicht noch einmal vorangestellt werden.

Beispiel:

```
class MyExplicitClass
{
    public:
        explicit MyExplicitClass(int i);
    private:
        int m_nID;
};
MyExplicitClass::MyExplicitClass(int i) : m_nID(i) {}

class MyClass
{
    public:
        MyClass(int i) : m_nID(i) {}
        MyClass(double d) : m_nID((int) d) {}
    private:
        int m_nID;
};

int main()
{
    MyExplicitClass ObjA(0);
    // ObjA = 7; //-> kompiliert nicht
    MyClass ObjB(0);
    ObjB = 8.8;
    ObjB = 4;
    return 0;
}
```

3.5 Zuweisungs-Operator

Der Operator = wird überschrieben:

```
class MyClass
{
    public:
        MyClass& operator=(const MyClass& Obj);
        ...
};
```

Beispiel:

```
class MyAbstractBase
{
    public:
        virtual ~MyAbstractBase() {} //Hack: nicht wirkll.inline
        virtual void Meth1() = 0; //rein virtuell
        void Meth2() { printf("Meth2()\n"); }
};

class MyAbstractMixinBase
{
    public:
        virtual void Meth2() = 0;
        virtual void Meth3() = 0;
        virtual void Meth4() = 0;
};
```

3.7 Default-Argumente

Um Default-Argumente zu definieren, setzt man in der Deklaration für alle Argumente ab einem bestimmten Argument einen Default-Wert an. Wenn man bspw. das dritte Argument per Default festlegt, dann muss auch das vierte, fünfte und alle weiteren per Default belegt werden:

```
class MyClass
{
    public:
        MyClass(int Prm1 = 0, int Prm2 = 1);
        void SetObjVal(int Prm1, int Prm2, int Prm3 = 0, int Prm4 = 0);
};
```

Wenn nun beim Aufruf nur der erste Teil der Argumente (mindestens bis zum Anfang der Default-Argumente) angegeben wird, dann wird der Rest der Argumente mit den Default-Werten gespeist.

3.8 Unspezifizierte Anzahl von Argumenten

Hat man zum Zeitpunkt des Programmierens eine unspezifizierte Anzahl von Argumenten für eine Funktion, dann kann man dies mit `...` anzeigen. Beispiel:

```
void format_string(string& strOUT, const char* szIN, ...)
```

```
{
}
```

Nun ist der Programmierer selbst dafür verantwortlich, die folgenden Argumente in ihrer Art und Anzahl zu unterscheiden und auszuwerten.

4.8 Ungarische Notation verwenden

Die ungarische Notation von Microsoft-Programmierer Charles Simonyi ist ein wirklich gutes Werkzeug, um den Überblick über die Variablen bei komplexen Projekten zu behalten. Die ungarische Notation kennzeichnet alle Variablen-Namen mit dem Typ, indem sie ein Präfix vergibt:

Stufe 1: Präfix unmittelbar vor dem Namen:

<code>bBoolean</code>	<code>bool</code> , <code>BOOL</code>	→ true, TRUE oder false, FALSE
<code>cName</code>	<code>char</code>	→ 8 Bit mit Vorz., -128...+127, 0..127 = ASCII-Std.
<code>nName</code>	<code>short</code>	→ 16 Bit mit Vorz., -32768...+32767
<code>lName</code>	<code>long</code>	→ 32 Bit mit Vorz., -2147483648...+2147483647
<code>iName, nName</code>	<code>int</code>	→ auf 16 Bit Betriebssystem <code>short</code> , auf 32 Bit <code>long</code>
<code>byName</code>	<code>unsigned char</code>	→ 1 Byte, 0...255, 0x0...0xFF
<code>wName</code>	<code>unsigned short</code>	→ 2 Byte, 0...65535, 0x0...0xFFFF
<code>dwName</code>	<code>unsigned long</code>	→ 4 Byte, 0...4294967295, 0x0...0xFFFFFFFF
<code>uiName, nName</code>	<code>unsigned int</code>	→ auf 16 Bit Betriebssystem <code>unsigned short</code> auf 32 Bit Betriebssystem <code>unsigned long</code>
<code>sName</code>	<code>char[]</code>	→ Zeichenkette, Vektor von <code>char</code> -Elementen
<code>szName</code>	<code>char[]</code>	→ Zeichenkette, die mit 0x00 endet (zeroterminated)
<code>strName</code>	<code>string</code>	→ string-Object (STL)
<code>fName</code>	<code>float</code>	→ 32 Bit Fließkomma, 7 Stellen
<code>dName</code>	<code>double</code>	→ 64 Bit Fließkomma, 15 Stellen
<code>eName</code>	<code>enum</code>	→ Aufzählung von <code>int</code>
<code>vName</code>	<code>void</code>	→ kann alles sein

Stufe 2: Präfix vor dem Präfix von Stufe 1:

<code>a...</code>	Array (Vektor, Matrix), Bsp.: <code>int anNumbers[256]</code> ;
<code>p...</code>	Pointer, Bsp.: <code>MyClass* pObj</code> ;
<code>sp...</code>	Smart-Pointer, Bsp.: <code>MyClassPtr spObj</code> ;

Stufe 3: Präfix vor dem Präfix von Stufe 2:

<code>g...</code>	Globale Variable
<code>m...</code>	Member-Variablen

Man sollte aber auch sonst auf einleuchtende Präfixe zurückgreifen:

```
list<MyClass>    listObjs;
set<MyClass>    setObjs;
```

Die Klassen und Strukturen eines Projektes sollte man sinnvoll bezeichnen und **nur bei Verkauf als Bibliothek an Dritte** ein herstellerbezeichnendes Präfix davorstellen (Bsp.: `class YString`). Nutzt man hingegen eine fremde Bibliothek, dann sollte deren Hersteller ebenfalls diese Regel beherzigen (Bsp.: Die Bibliothek 'MFC' von Microsoft benutzt das 'C': `class CString`). Ein Präfix für Klassennamen sollte jedoch wegen der Wiederverwendbarkeit von Code in anderen Projekten **nie projektabhängig** sein bzw. den Projektnamen beinhalten.

Hier das erste allgemeine Beispiel:

```
#include <stdio.h>
#include <string>
using namespace std;
int main()
{
    string strText("Hello"); //-> "Hello"

    //Umwandeln in char*:
    unsigned long dwLen = strText.size() + 1;
    char* szText = new char[dwLen];
    strcpy(szText, strText.c_str());
    delete[] szText;
    szText = NULL;

    //" " anhängen:
    strText.append(" ");

    //strText2 anhängen:
    string strText2("World");
    strText += strText2;

    //Ab Zeichen 6 "--- " einfügen:
    strText.insert(6, "--- ");

    //Ab Zeichen 8 strInsertStr einfügen:
    string strInsertStr("*-");
    strText.insert(8, strInsertStr.begin());

    //5 mal '>' davor:
    strText.insert(strText.begin(), 5, '>');

    //Ab Zeichen 5 " #" einfügen:
    strText.insert(5, " # ");

    //lNum dezimal ab Zeichen 7 einfügen:
    long lNum = 10101;
    char szNum[256];
    sprintf(szNum, "%ld", lNum);
    strText.insert(7, szNum);

    //Am Anfang "#0x :" einfügen:
    strText.insert(0, "#0x: ");

    //lNum hexadezimal ab Zeichen 3 einfügen:
    lNum = 65535;
    sprintf(szNum, "%X", lNum);
    strText.insert(3, szNum);

    //HEX-Code eines Zeichens ermitteln:
    char szSpecial[2] = "é"; //Sonderzeichen (oberhalb 0x7F)
    sprintf(szNum, "<%02Xh>",
            (unsigned char) szSpecial[0]);
    //Visual C++: benötigt cast nach 'unsigned char'
    return 0;
}
```

```

int main()
{
    string strText1("Nice ");
    string strText2("This day");

    //3 Zeichen von strText2 anhängen, beginnend beim Index 5:
    strText1.append(strText2,5,3);

    printf("%s\n",strText1);
    return 0;
}

```

5.2.7 Konfigurationsdateien parsen mit `string::compare()` und `string::copy()`

Um die Anwendung von `string::compare()` und `string::copy()` zu zeigen, eignet sich das Beispiel einer einfachen Konfigurationsdatei, wie sie z.B. unter Linux verwendet wird, um den ersten Ethernet-Adapter zu konfigurieren (zu finden unter `/etc/sysconfig/network-scripts/ifcfg-eth0`). Die Datei 'ifcfg-eth0' hat etwa folgendes Format:

```

DEVICE=eth0
BOOTPROTO=static
IPADDR=175.44.5.40
BROADCAST=172.26.2.255
NETMASK=255.255.0.0
GATEWAY=175.44.0.1
ONBOOT=yes

```

Es werden also die Namen der Parameter linksbündig mit Gleichheitszeichen in die Datei geschrieben und der Wert wird ohne Lücke (SPACE) drangehängt. Die Datei lässt sich nun einfach parsen:

```

#include <stdio.h>
#include <string>
using namespace std;
bool GetLine(FILE* pFile,string& strLine)
{
    strLine = "";
    char szChar[2] = "?";
    while((szChar[0] = (char) fgetc(pFile)) != EOF)
    {
        if(szChar[0] == 0x0D) //CR ("\r")
            continue;
        if(szChar[0] == 0x0A) //LF ("\n")
            break;
        strLine += szChar;
    }
    if(szChar[0] == EOF)
    {
        if(strLine.empty())
            return false;
    }
    return true;
}

```

6. Zeitermittlung

6.1 Weltweit eindeutiger Timestamp (GMT), Jahr-2038

Wenn man Daten über den gesamten Globus austauscht, dann ist es manchmal wichtig eine weltweit eindeutige Zeit zu verwenden. Zum einen, um Abläufe zu synchronisieren, und zum anderen, um Zeitangaben unabhängig vom Ort zu ermitteln bzw. auszuwerten. Die GMT (**Greenwich Mean Time**) ist eine solche Zeit, die mit Hilfe der System-Uhr ermittelt wird. Um eine genaue Zeitangabe zu erhalten ist es wichtig, eine Synchronisierung der eigenen System-Uhr mit einer Normal-Uhrzeit vorzunehmen. Dazu kann man das Network Time Protocol (NTP) verwenden, d.h. man startet einen NTP-Client als Hintergrund-Prozess, der bei einem NTP-Server (Daemon) regelmäßig die Zeit erfragt und die eigene System-Uhr nachstellt.

Beispiel:

```
#include <stdio.h>
#include <time.h>

int main()
{
    long lTime = time(NULL);
    tm tmGMT = *gmtime(&lTime);

    char szTimeStamp[128] = "";
    sprintf(    szTimeStamp,
                "%04d-%02d-%02d %02d:%02d:%02d",
                (tmGMT.tm_year + 1900),
                tmGMT.tm_mon + 1,
                tmGMT.tm_mday,
                tmGMT.tm_hour,
                tmGMT.tm_min,
                tmGMT.tm_sec);

    printf("Timestamp: %s\r\n",szTimeStamp);

    return 0;
}
```

Problem:

`time(NULL)` liefert die Zeit in **Sekunden, gezählt vom 1. Januar 1970, 00:00 Uhr**. Leider wird der Wert zudem noch als **long** (-2147483648...+2147483647) zurückgeliefert, was bedeutet, dass es nach 68 Jahren (+2147483647 / (365.25 * 86400)), also **im Jahre 2038** zu einem Überlauf kommt.

Dies ist, nebenbei bemerkt, auch das Jahr in dem die 16-Bit-Zähler (`unsigned short`) des Modifizierten Julianischen Datums (**MJD; Tage, gezählt vom 17. November 1858**) überlaufen, was ich in der Tat als seltsamen Zufall empfinde. 16-Bit-MJD wird z.B. in den MPEG-2-Streams von DVB (Digital Video Broadcast = Digitales Fernsehen) zur Spei-

7. Konstantes

7.1 const-Zeiger (C-Funktionen)

C kennt im Gegensatz zu C++ keine Referenzparameter und benutzt Zeiger als Argumente, wenn eine Funktion einen Parameter manipulieren soll. Um innerhalb einer C-Funktion einen Parameter manipulieren zu können, muss der Aufrufer die zu übergebende Variable mit dem **Operator & referenzieren**, d.h. der Operator & ermittelt die Speicheradresse und liefert sie zurück, weist sie also somit dem Zeigerparameter zu. Innerhalb der Funktion kann der Wert nun mit Hilfe des Operators * gelesen oder beschrieben werden, d.h. der Zeiger wird über den **Operator * dereferenziert**:

- **Zeiger auf Speicherstelle (Lesen/Beschreiben einer Variablen):**

Funktion:

```
void f(int* pParam)
{
    ++(*pParam); //dereferenzieren des Zeigers
}
```

Aufruf:

```
int nValue = 0;
f(&nValue); //referenzieren der Variablen
```

- **Zeiger auf Zeiger (Lesen/Beschreiben eines Zeigers):**

Funktion:

```
void f(int** ppParam)
{
    static int anValues[10];
    *ppParam = &anValues[0]; //deref. d.äusseren Zeigers
}
```

Aufruf:

```
int* pnValues = NULL;
f(&pnValues); //referenzieren des Zeigers
```

Mittels **const** kann man verhindern, dass verschiedene Sachen manipuliert werden:

```
void f(const char* p){...} //konstante Daten
void f(char* const p){...} //konstanter Zeiger
void f(const char* const p){...} //konst.Daten & konst.Zeiger
```

Zu allem Überfluss ist auch noch eine andere Schreibweise erlaubt:

```
const char* p                       $\longleftrightarrow$                       char const* p
```

identisch

Regel, die immer gilt: **Was links von * steht, hält die Daten konstant!**

```

void OutputDebugString(const char* szStr)
{
    ...; //Ausgabe in ein DEBUG-Fenster des Debuggers
}

int main()
{
    TRACE("Test\n");
    return 0;
}

```

- **Variablennamen zusammenbauen**

```

#define GET_WORD(w) (unsigned short) \
    ( ((unsigned short) ##w##_HI) << 8) + ##w##_LO)

int main()
{
    unsigned char Value_HI = 0xAA;
    unsigned char Value_LO = 0x55;
    unsigned short wValue = GET_WORD(Value);
    printf("wValue = 0x%04X\r\n", wValue);
    return 0;
}

```

- **Funktionsnamen tracen**

```

#define CALL_FUNC(Function) \
    (##Function##); \
    printf("Function [%s] was called\r\n", #Function)

int Func1(int i)
{
    return i;
}
char Func2(char c)
{
    return c;
}

int main()
{
    int i = CALL_FUNC(Func1(3));
    printf("i = %ld\r\n", i);
    char c = CALL_FUNC(Func2('A'));
    printf("c = %c\r\n", c);
    return 0;
}

```

9. Referenz statt Zeiger (Zeiger für C-Interface)

Wo ein Zeiger ist, ist in der Regel auch ein **new** (bzw. `new[]`) und hoffentlich ein **delete** (bzw. `delete[]`). Wenn Zeiger an Funktionen übergeben werden, d.h. es existiert mehr als eine Kopie des Zeigers, dann taucht automatisch die zu lösende Frage auf, wer das `delete` ausführen soll und vor allem wann er dies zu tun hat oder tun darf. Wenn nämlich der eine `delete` auf den Zeiger anwendet und der andere noch mit dem Zeiger arbeitet, entsteht logischerweise ein Problem.

Diese Probleme sollten mit C++ der Vergangenheit angehören, denn es gibt ja **Referenzen** (und die STL, welche das komplette Heap-Management kapselt, also alles, was mit `new` und `delete` zu tun hat). Hier ein paar wichtige Unterschiede zwischen Referenz und Zeiger:

- Auf eine Referenz kann man kein `delete` anwenden
- Referenzen zeigen immer auf ein gültiges (nicht gelöscht) Objekt
- Eine Referenz zeigt immer nur auf ein und dasselbe Objekt
- Der Wert `NULL` für eine Referenz ist nicht definiert

Beispiel:

```
MyClass Object;
MyClass& Obj1;           //-> Compiler-Fehler
MyClass& Obj2 = Object; //-> ok
```

Man sollte **nie einer Referenz einen dereferenzierten Zeiger zuweisen**, da der Zeiger `NULL` sein kann und eine Dereferenzierung von `NULL` nicht definiert ist..

Beispiel:

```
void g(MyClass& Obj)
{
    MyClass LocalObj = Obj;
}
int main()
{
    MyClass* pObj = NULL;
    g(*pObj); //-> access violation zur Laufzeit
    return 0;
}
```

Die Besonderheit, dass eine Referenz zur Laufzeit immer auf ein gültiges Objekt zeigen muss, erzwingt Folgendes: Eine **Referenz, welche Member-Variable ist**, muss immer bereits in der **Initialisierungsliste** des Konstruktors mit einem gültigen Wert belegt werden.

10.3 Überladen innerhalb einer Klasse vs. über Klasse hinweg

10.3.1 Allgemeines

Es ist etwas anderes, ob man innerhalb einer Klasse eine Methode überlädt oder ob man eine Methode der Basisklasse überlädt.

Überladen innerhalb einer Klasse:

Wenn man innerhalb einer Klasse eine Methode überlädt, so entsteht eine Koexistenz von mehreren Funktionen gleichen Namens, sobald sich ihre **Parameter hinsichtlich Art und/oder Anzahl** unterscheiden.

Überladen einer Methode der Basisklasse:

Wenn man eine Methode der Basisklasse überlädt, dann geschieht dies ohne Rücksicht auf die Parameter. Es genügt, den gleichen Funktionsnamen zu benutzen. Das heißt aber auch, dass eine überladene Methode der Basis folgendermaßen geändert werden kann:

- Die Art/Anzahl der Argumente kann sich ändern
- Der return-Wert kann sich ändern

Beispiel:

```
class MyBase
{
    public:
        virtual ~MyBase() {} //Hack: nicht wirklich inline
        virtual int Func(int i,int j);
};
int MyBase::Func(int i,int j) { return (i + j); }

class MyClass : public MyBase
{
    public:
        virtual ~MyClass() {} //Hack: nicht wirklich inline
        virtual double Func(const double& d);
};
double MyClass::Func(const double& d) { return d; }

int main()
{
    MyClass Obj;
    // i = Obj.Func(6,4); //-> kompiliert nicht!
    double d = Obj.Func(8.0);

    return 0;
}
```

Man sollte ein solches Vorgehen jedoch vermeiden. Der Compiler bringt hier in der Regel eine Warnung.

10.3.2 Nie Zeiger-Argument mit Wert-Argument überladen

Wenn man eine Überladung vornimmt, wobei die **Art** der Argumente der Funktionen unterschiedlich ist (was man vermeiden soll), dann sollte man nicht zugleich Zeiger und Werte für ein Argument zulassen.

Beispiel:

```
void MyClass::Func(int i);  
void MyClass::Func(MyClass* pObj);
```

→ Func(0) ruft **immer** Func(int i=0) auf, obwohl auch der NULL-Zeiger gemeint sein könnte, also Func(MyClass* pObj=NULL).

10.4 return: Referenz auf *this vs. Wertrückgabe

Wie man das Ergebnis einer Methode zurückgibt, hängt davon ab, ob ein lokal in der Methode erzeugtes Objekt oder das Objekt, zu dem die Methode gehört, zurückgeliefert werden soll.

10.4.1 Lokal erzeugtes Objekt zurückliefern: Rückgabe eines Wertes

Will man ein Objekt zurückliefern, welches man in einer Methode erzeugt hat, kann man keine Referenz (und keinen Zeiger) darauf zurückgeben, denn das lokal erzeugte Objekt wird ja wieder zerstört, wenn man die Funktion verlässt. Man gibt hier einen **Wert** oder einen **const-Wert** zurück (also eine Kopie des lokal erzeugten Objektes).

Beispiel:

```
class MyClass  
{  
    public:  
        double MyClass::Calculate(const double& dInput)  
        {  
            double dReturn = dInput * 10;  
            return dReturn; //-> Wert  
        }  
};  
  
int main()  
{  
    MyClass Obj;  
    double d = Obj.Calculate(7.0);  
    return 0;  
}
```

12.6 Heap-Speicher als Shared Memory

Wenn bereits ein Puffer (pBuf) existiert, dann kann man beliebig oft beliebig verschiedene Objekte dort platzieren, solange diese nicht größer als der reservierte Speicherbereich sind.

Beispiel:

```
#include <new.h>
#include <list>
using namespace std;

class MyString
{
public:
    MyString(const char* const szStr)
    {
        strcpy(m_szStr,szStr);
    }
    ~MyString() { printf("Destruction\n"); }
    char& operator[](int pos)
    {
        return m_szStr[pos];
    }
    void PrintIt() { printf("%s\n",m_szStr); }
protected:
    char m_szStr[256];
};

int main()
{
    //Heap als Shared-Memory reservieren:
    const size_t BYTE_SIZE = 1024 * 1024; //1 MB
    void* pSharedMemory = operator new(BYTE_SIZE);
    void* pMem = pSharedMemory;

    //Shared-Memory als String-Objekt nutzen:
    MyString* pStr = new (pMem) MyString("Hello");

    //Shared-Memory als Liste mit Integern benutzen:
    list<int>* plistInts = new (pMem) list<int>;
    plistInts->push_back(1);
    plistInts->push_back(2);
    plistInts->push_back(3);

    //Allokierten Heap der Liste wieder freigeben:
    plistInts->clear();

    //Shared-Memory wieder als String-Objekt nutzen:
    pStr = new (pMem) MyString("World");

    //Shared-Memory wieder freigeben:
    operator delete(pSharedMemory);

    return 0;
}
```

14.4.2 Referenz auf **this* zurückliefern

Der Zuweisungs-Operator muss eine Referenz auf **this* zurückliefern!

Gründe:

- Verkettung muss möglich sein:

Beispiel: `x = y = z = 0;`

→ Der Rückgabewert des einen Operators ist gleich dem Argument des anderen:

```
z ← 0
y ← z
x ← y
```

- Zeigerkonflikt muss vermieden werden:

Referenz auf das Argument ist nicht erlaubt, da sonst der Empfänger nicht eine Referenz auf das Objekt mit dem zugewiesenen Wert bekommt, sondern eine Referenz auf das Objekt, von dem die Werte übernommen wurden.

14.4.3 Alle Member-Variablen belegen

Es müssen die Werte **aller** Member-Variablen zugewiesen werden, **auch die der Basisklassen!**

Beispiel:

```
class MyBase
{
    public:
        MyBase(const int nID = 0) : m_nID(nID) {}
        virtual ~MyBase() {} //Hack: nicht wirklich inline
        void SetID(int nID) { m_nID = nID; }
        int GetID() const { return m_nID; }
    private:
        int m_nID;
};
```

```

class MyClass : public MyBase
{
    public:
        MyClass(const int nID = 0, const int nValue = 0)
            : MyBase(nID), m_nValue(nValue) {}
        virtual ~MyClass() {}
        MyClass& operator=(const MyClass& Obj)
        {
            if(this == &Obj)
                return *this;
            SetID(Obj.GetID()); //Basisklasse
            SetValue(Obj.GetValue()); //Eigene Member
            return *this;
        }
        void SetValue(int nValue) { m_nValue = nValue; }
        int GetValue() const { return m_nValue; }
    private:
        int m_nValue;
};

int main()
{
    MyClass Obj1(4,3);
    MyClass Obj2(6);
    printf(
        "Obj2: ID=%ld Value=%ld\r\n", Obj2.GetID(), Obj2.GetValue());
    Obj2 = Obj1;
    printf(
        "Obj2: ID=%ld Value=%ld\r\n", Obj2.GetID(), Obj2.GetValue());
    return 0;
}

```

14.5 Indizierter Zugriff per operator[]()

Der operator[] () muss als Ergebnis immer eine **Referenz** zurückliefern, denn diese wird benötigt um nicht nur lesend, sondern **auch schreibend** auf die per Index adressierte Speicherstelle zugreifen zu können.

Beispiel:

```

#include <string.h> //memset
class MyIntArr
{
    public:
        MyIntArr()
        {
            memset(m_anInt, 0, sizeof(m_anInt));
        }
        int& MyIntArr::operator[](unsigned long pos)
        {
            if(pos > 255)
                return m_anInt[0];
            return m_anInt[pos];
        }
}

```

```

const int* GetArray() const
{
    return m_anInt;
}
MyIntArr& operator=(const MyIntArr& Obj)
{
    if(this == &Obj)
        return *this;
    const int* panInt = Obj.GetArray();
    for(int i = 0;i < 256;++i)
        m_anInt[i] = *(panInt++);
    return *this;
}
private:
    int m_anInt[256];
};
int main()
{
    MyIntArr A;
    printf("A[3]=%ld\r\n",A[3]);
    A[3] = 5;
    printf("A[3]=%ld\r\n",A[3]);
    return 0;
}

```

14.6 Virtuelle Clone()-Funktion: Heap-Kopie über pBase

Mittels virtueller Clone()-Funktion kann man Heap-Kopien über den Basisklassen-Zeiger machen. Man kann also eine Funktion implementieren, die über den Basisklassen-Zeiger (statischer Typ) operiert, aber jederzeit eine Kopie des dynamischen Typs erzeugen kann.

Beispiel:

```

#include <list>
#include <string>
using namespace std;

class MyBase
{
public:
    virtual ~MyBase() {} //Hack: nicht wirkll.inline
    virtual MyBase* Clone() const = 0;
    virtual string GetType() const;
};

string MyBase::GetType() const
{
    return string("Base");
}

```

15. Richtiges Vererbungs-Konzept

15.1 Allgemeines

15.1.1 Nie von (nicht-abstrakten) Klassen ohne virtuellen Destruktor erben

Da im Destruktor der abgeleiteten Klasse Heap-Speicher freigegeben werden kann, sollte der Destruktor der Basisklasse virtuell sein und somit die Destruktion an die abgeleitete Klasse weiterleiten.

Beispiel:

```
#include <string.h> //memset
class MyBaseArray
{
    public:
        MyBaseArray() { memset(m_anInt,0,sizeof(m_anInt)); }
        virtual ~MyBaseArray() {} //Hack: nicht wirklich inline
        virtual MyBaseArray* GetHeapArrayExt();
    protected:
        int m_anInt[256];
};
MyBaseArray* MyBaseArray::GetHeapArrayExt()
{
    return NULL;
}

class MyArray : public MyBaseArray
{
    public:
        MyArray() : m_pHeapArrayExt(new MyBaseArray()) {}
        virtual ~MyArray();
        virtual MyBaseArray* GetHeapArrayExt();
    private:
        MyBaseArray* m_pHeapArrayExt; //Heap-Erweiterung
};
MyArray::~MyArray()
{
    delete m_pHeapArrayExt;
}
MyBaseArray* MyArray::GetHeapArrayExt()
{
    return m_pHeapArrayExt;
}

int main()
{
    MyBaseArray* pHeapArray = new MyArray();
    MyBaseArray* pHeapArrayExt = pHeapArray->GetHeapArrayExt();
    delete pHeapArray; //nur ok, weil Basisdestruitor virtuell ist
    return 0;
}
```

15.1.3 Statischer/dynamischer Typ und statische/dynamische Bindung

Der Compiler bindet **nicht-virtuelle Funktionen** statisch, d.h. ihr Code steht bereits nach der Kompilierung fest. **Virtuelle Methoden** werden dynamisch gebunden. Das bedeutet, dass erst zur Laufzeit feststeht, welcher Code zu verwenden ist (vftable).

Als **statischen Typ** eines Zeigers bezeichnet man den Typ, auf den er bei der Definition zeigt:

```
MyBase* pObj = NULL; //statischer Typ: MyBase
```

Als **dynamischen Typ** bezeichnet man den Typ des Objektes, auf den der Zeiger zur Laufzeit zeigt:

```
pObj = new MyClass(); //dynamischer Typ: MyClass
```

Der Aufruf **virtueller Funktionen** wird über den **vfptr** an die **vftable** geleitet und von da an den dynamischen Typ weitergegeben → **dynamische Bindung**. Der Aufruf **nicht-virtueller Funktionen** wird an den **statischen Typ** weitergeleitet → **statische Bindung**.

Beachte: Default-Argumente werden immer der statischen Bindung entnommen!

Beispiel:

```
class MyBase
{
    public:
        MyBase() {}
        virtual ~MyBase() {} //Hack: nicht wirklich inline
        void PrintStaticType() { printf("MyBase\n"); }
        virtual void PrintDynamicType();
};
void MyBase::PrintDynamicType() { printf("MyBase\n"); }

class MyClass : public MyBase
{
    public:
        MyClass() {}
        virtual ~MyClass() {} //Hack: nicht wirklich inline
        void PrintStaticType() { printf("MyClass\n"); }
        virtual void PrintDynamicType();
};
void MyClass::PrintDynamicType() { printf("MyClass\n"); }

int main()
{
    MyBase* pObj = new MyClass();
    pObj->PrintStaticType();
    pObj->PrintDynamicType();
    return 0;
}
```

- Was wird wie geerbt?

Generell erbt man **immer alle public- und protected Member** der Basisklasse. Das Schlüsselwort bei der Vererbung (public, protected oder private) gibt jedoch die **höchst zulässige Öffentlichkeit** vor:

- o **public-Vererbung:**

```
class MyClass : public MyBase
{
    ...
};
```

→ MyClass erbt alle **public-** und **protected-**Member von MyBase, und zwar als **public-** und **protected-**Member (alles bleibt unverändert)

- o **protected-Vererbung:**

```
class MyClass : protected MyBase
{
    ...
};
```

→ MyClass erbt **public-** und **protected-**Member von MyBase, und zwar als **protected-**Member (public wird also zu protected)

- o **private-Vererbung:**

```
class MyClass : private MyBase
{
    ...
};
```

→ MyClass erbt **public-** und **protected-**Member von MyBase, und zwar als **private-**Member (alles wird zu private)

Beispiel:

```
class MyBase
{
    public:
        MyBase() {}
        virtual ~MyBase() {} //Hack: nicht wirklich inline
        void PrintNo(int nNo) { printf("No = %d\n",nNo); }
};
class MyClass : protected MyBase
{
    public:
        MyClass() {}
        virtual ~MyClass() {} //Hack: nicht wirklich inline
        void Print(int No) { PrintNo(No); }
};
```

15.4 Composition statt multiple inheritance

Wenn man die Eigenschaften verschiedener Klassen in einem Objekt vereinen will, dann sollte man diese Klassen nicht als Basisklassen einer Mehrfach-Erbung (multiple inheritance) ansetzen, denn Mehrfach-Erbung ist allein wegen der Mehrdeutigkeit der Namen schon kritisch. Statt dessen sollte man in die Klasse mit den vereinten Eigenschaften **Zeiger auf die Implementierungs-Klassen** deklarieren. Dieses Konzept nennt man **Composition** (Zusammenfügung).

Beispiel:

```
//----- MyClass.h: -----
class MyImpl1;
class MyImpl2;
class MyClass
{
    public:
        MyClass()
            : m_pImpl1(new MyImpl1), m_pImpl2(new MyImpl2) {}
        ~MyClass();
        void Do();
        void Verify();
    private:
        MyImpl1* m_pImpl1;
        MyImpl2* m_pImpl2;
};

//----- MyImpl1.h: -----
class MyImpl1
{
    public:
        MyImpl1() {}
        void Do();
};

//----- MyImpl1.cpp: -----
void MyImpl1::Do() { printf("MyImpl1::Do() done!\n"); }

//----- MyImpl2.h: -----
class MyImpl2
{
    public:
        MyImpl2() {}
        void Verify();
};

//----- MyImpl2.cpp: -----
void MyImpl2::Verify() { printf("MyImpl2::Verify() done!\n"); }

//----- MyClass.cpp: -----
MyClass::~MyClass()
{
    if(m_pImpl1)
        delete m_pImpl1;
    if(m_pImpl2)
        delete m_pImpl2;
}
```

Beispiel:

```
class A
{
    public:
        explicit A(int nID = 0) : m_nID(nID) {}
        ~A() {}
        void Calculate() { printf("A::Calculate()\n"); }
        void Read() const { printf("A::Read()\n"); }
        void SetID(int nID) { m_nID = nID; printf("A::SetID()\n"); }
        int GetID() const { printf("A::GetID()\n"); return m_nID; }
    private:
        int m_nID;
};
class B : virtual public A
{
    public:
        explicit B(int nID = 0) : A(nID) {}
        virtual ~B() {}
        virtual void Calculate() { printf("B::Calculate()\n"); }
        void AlgorithmB() { printf("B::AlgorithmB()\n"); }
};
class C : virtual public A
{
    public:
        explicit C(int nID = 0) : A(nID) {}
        virtual ~C() {}
        virtual void Read() { printf("C::Read()\n"); }
        void AlgorithmC() { printf("C::AlgorithmC()\n"); }
};
class D : public B, public C
{
    public:
        explicit D(int nID = 0) { SetID(nID); }
};
int main()
{
    D Obj(8);
    int i = Obj.GetID(); //wird direkt von A übernommen
    Obj.AlgorithmB(); //wird direkt von B übernommen
    Obj.AlgorithmC(); //wird direkt von C übernommen
    Obj.Calculate(); //wird nach überschreiben von B übernommen
    Obj.Read(); //wird nach überschreiben von C übernommen
    return i;
}
```

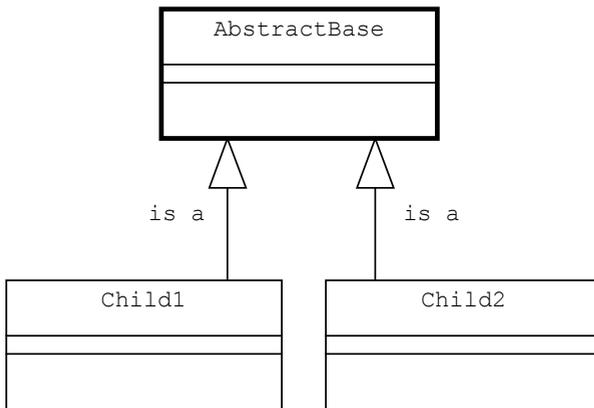
Probleme, die hierbei auftreten:

- Beim Entwurf von A, B und C kann man noch nicht wissen, dass später mal eine Diamant-Struktur entstehen wird, d.h. dass für das Erben von A später mal **virtuelle** Vererbung erforderlich wird. Deshalb wird in der Regel beim Entwurf von A nicht berücksichtigt, dass A **keine virtuellen Funktionen** enthalten darf. Außerdem wird B und C in der Regel nicht virtuell von A abgeleitet vorliegen.
- **Eindeutigkeit** der Methoden der virtuellen Basisklasse A ist nur gegeben, wenn B nur Methoden überschreibt, die C nicht überschreibt und umgekehrt. Wenn also Calculate() durch B und durch C überschrieben wird, dann meldet der Compiler einen Fehler (ambiguous).

15.9 Zuweisungen nur zwischen gleichen Child-Typen zulassen

Damit der Inhalt eines Childs (Spezialisierung einer Basisklasse über "is-a"-Vererbung) nur dann einem anderen Child zugewiesen wird, wenn beide vom gleichen Typ sind, muss jede Child-Klasse einen eigenen Zuweisungsoperator definieren und die gemeinsame Basisklasse muss diesen Operator verstecken.

Beispiel:



```
class AbstractBase
{
    public:
        virtual ~AbstractBase() {} //Hack: nicht wirklich inline
        virtual void Do() = 0;
        void SetID(int nID) { m_nID = nID; }
        int GetID() const { return m_nID; }
    private:
        //Versteckter Zuweisungsoperator:
        AbstractBase& operator=(const AbstractBase& Obj);

        int m_nID;
};

class Child1 : public AbstractBase
{
    public:
        Child1(int nID = 1) { SetID(nID); }
        virtual ~Child1() {}
        virtual void Do();
        Child1& operator=(const Child1& Obj)
        {
            if(this == &Obj)
                return *this;
            SetID(Obj.GetID());
            return *this;
        }
};

void Child1::Do() { printf("Child1::Do() / ID = %d\n",GetID()); }
```

```

class Child2 : public AbstractBase
{
    public:
        Child2(int nID = 2) { SetID(nID); }
        virtual ~Child2() {}
        virtual void Do();
        Child2& operator=(const Child2& Obj)
        {
            if(this == &Obj)
                return *this;
            SetID(Obj.GetID());
            return *this;
        }
};

void Child2::Do() { printf("Child2::Do() / ID = %d\n",GetID()); }

void Copy(const AbstractBase& Src,AbstractBase& Dst)
{
    Dst = Src; //-> baut nicht, da der operator versteckt ist
}

int main()
{
    Child1 Obj_11(11);
    Child1 Obj_12(12);
    Obj_11.Do();
    Obj_12.Do();

    Copy(Obj_11,Obj_12);
    Obj_12.Do();

    Child2 Obj_21(21);
    Copy(Obj_11,Obj_21); //-> nicht erlaubte Zuweisung
    Obj_21.Do();
    return 0;
}

```

16. Nutzer einer Klasse von Änderungen entkoppeln

16.1 Allgemeines

Wenn sich an der **Anzahl** oder der **Aufruf-Konvention** der **private-Methoden** (Funktionen und Sub-Funktionen der internen Implementierung) etwas ändert, dann bekommt der Benutzer der **public-Methoden** das gut zu spüren, obwohl er die **private-Methoden** gar nicht aufrufen kann. Der Grund hierfür ist, dass die durch `#include` eingebundene Header-Datei `*.h` sich ändert. Dies zieht also nach sich, dass **alle *.cpp-Module, die diese Header-Datei einbinden, neu kompiliert werden müssen**. Da dies ein sehr unschöner Effekt ist, hat man verschiedene Verfahren und Regeln entwickelt, die so etwas vermeiden.

16.2 Header-Dateien: Forward-Deklaration statt #include

In Header-Dateien (Klassendeklarationen) hat man oftmals die Möglichkeit `#include` zu vermeiden. Der Hintergrund ist, dass man eine fremde Klassendeklaration (also `#include`) nur dann benötigt, wenn man Speicher für ein fremdes Objekt allokiert, d.h. wenn man ein Objekt einer fremden Klasse als Member-Variable benutzt oder Methoden der Klasse über Zeiger oder Referenzen aufruft. Ist dies in der Klassendeklaration nicht der Fall, hat man also bestenfalls **Zeiger** oder **Referenzen** auf Objekte anderer Klassen definiert (ohne damit auf deren Methoden zuzugreifen), dann muss man lediglich den Namen der Klasse bekannt machen (Forward-Deklaration).

In Header-Dateien sollte man also versuchen `#include` durch die Forward-Deklaration zu ersetzen:

- Nicht auf Referenz-Parameter zugreifen (Code in die Implementierung verlegen)
- Keine Allokierung von Heap-Speicher (`new`) für Zeiger (Code in die Implementierung verlegen)
- Komplexe Implementierungen per Zeiger aufnehmen

Beispiel:

```
class MyMember; //Forward-Deklaration
class MyImpl;   //Forward-Deklaration

class MyClass
{
public:
    MyClass() : m_pImpl(new MyImpl) {}
    void Do(const MyMember& Obj); //hier kein Zugriff auf Obj
private:
    MyImpl* m_pImpl; //Zugriff auf Implementierung per Zeiger
};
```

16.3 Delegation bzw. Aggregation

Ein Konzept, welches zwar den Aufbau der Software maßgeblich ändert, aber die `#include`-Anweisung vermeidet und damit den Nutzer einer Klasse von Änderungen der `private`-Methoden (also der internen Implementierung) entkoppelt, ist die **Implementierung per Letter-Klasse, auf die es dann einen Zeiger in der eigentlichen Klasse gibt**. Die eigentliche Klasse (Envelope, Umschlag) hält also lediglich einen **Zeiger auf die Implementierung** (Letter, Brief) und beherbergt damit nicht die internen Implementierungs-Methoden hinter `private`.

Beispiel:

```
//*****  
// MyClass.h  
//*****  
  
class MyImpl; //Forward-Deklaration der Letter-Klasse  
  
class MyClass //Envelope-Klasse für MyImpl  
{  
public:  
    MyClass() : m_pImpl(new MyImpl) {}  
    ~MyClass();  
    void Meth1();  
    void Meth2();  
private:  
    MyImpl* m_pImpl; //Zugriff auf Implementierung per Zeiger  
};
```

Die **Envelope-Klasse** implementiert eine **Delegation** an die Implementierung, also die Letter-Klasse. Man kann auch sagen, sie **aggregiert** Letter-Klassen-Aufrufe:

```
//*****  
// MyClass.cpp  
//*****  
  
#include "MyClass.h" //Envelope-Klasse  
#include "MyImpl.h" //Letter-Klasse  
  
MyClass::~MyClass()  
{  
    if(m_pImpl)  
        delete m_pImpl;  
}  
void MyClass::Meth1()  
{  
    if(m_pImpl)  
        m_pImpl->Meth1(); //Delegation (Aggregation)  
}  
void MyClass::Meth2()  
{  
    if(m_pImpl)  
        m_pImpl->Meth2(); //Delegation (Aggregation)  
}
```

Die **Letter-Klasse** hingegen implementiert die Funktionalität und beinhaltet die gesamten Methoden (interne Funktionen und Sub-Funktionen), die dazu notwendig sind:

```
//*****
// MyImpl.h:
//*****

class MyImpl
{
    public:
        void Meth1();
        void Meth2();
    private: //interne Implementierung
        int Calculate1(int n);
        int Calculate2(int n);
        void Print(int n);
};

//*****
// MyImpl.cpp:
//*****

#include "MyImpl.h"

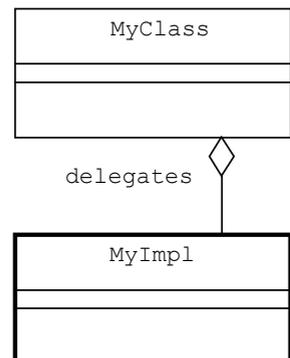
void MyImpl::Meth1()
{
    int i = Calculate1();
    Print(i);
}

void MyImpl::Meth2()
{
    int i = Calculate2();
    Print(i);
}

int MyImpl::Calculate1(int n)
{
    return n + 1;
}

int MyImpl::Calculate2(int n)
{
    return n + 2;
}

void MyImpl::Print(int n)
{
    printf("Calculated number: %d\n",n);
}
```



Beispiel mit dem Operator ++:

```
class MyClass
{
    public:
        MyClass(int nID = 0) : m_nID(nID) {}
        MyClass& operator++()           //-> Präfix ( ++Obj )
        {
            ++m_nID;
            return *this; //Rückgabe einer Referenz auf *this
        }
        const MyClass operator++(int)   //-> Postfix ( Obj++ )
        {
            MyClass Obj(m_nID); //temporäres Objekt
            ++m_nID;
            return Obj; //Rückgabe eines Objektes per Wert
        }
    private:
        int m_nID;
};

int main()
{
    MyClass Obj;
    ++Obj;    //-> Obj.operator++();
    Obj++;    //-> Obj.operator++(0);
    return 0;
}
```

Der **Postfix**-Operator muss einen **Wert** zurückgeben (keine Referenz auf *this), da er (wie man sieht) das Objekt für die Rückgabe nur lokal und temporär erzeugen kann. Er sollte aber immer einen **const-Wert zurückgeben**, damit folgendes **falsche Verhalten** verhindert wird:

```
/* const */ MyClass MyClass::operator++(int)    //-> Postfix (Obj++)
{
    MyClass Obj(m_nID); //temporäres Objekt
    ++m_nID;
    return Obj; //Rückgabe eines Objektes per Wert
}
```

→ Obj++++; würde zu Obj.operator++(0).operator++(0);

Dies würde bedeuten, dass **auf dem temporären Objekt der ersten Operation** nochmal ++ aufgerufen würde. Dies veränderte aber nicht den Wert des eigentlichen Objektes und bliebe daher ohne Auswirkung auf die beteiligten Objekte, und es wäre:

Obj2 = Obj1++++; identisch mit Obj2 = Obj1++;

was nur schwer einleuchtend ist.

→ **static_cast** kann stattdessen benutzt werden, wenn das Objekt **keinen vfptr** hat.

- Man sollte immer wenn ein cast schief gehen kann `dynamic_cast` verwenden und **nicht static_cast**. `dynamic_cast` ist zwar langsamer, dafür aber sicherer.

Beispiel für fehlschlagenden cast:

```
class MyBase
{
public:
    MyBase(int nID = 0) : m_nID(nID) {}
    virtual ~MyBase() {}
    void SetID(int nID) { m_nID = nID; }
    int GetID() const { return m_nID; }
private:
    int m_nID;
};

class MyChild : public MyBase
{
public:
    MyChild() {}
    virtual ~MyChild() {}
};

int main()
{
    //Upcast von Child-Objekt (dynamischer bzw.eigentlich.Typ)
    //zu Basis-Zeiger/-Referenz (statischer Typ)
    //-> Bei richtiger "is-a"-Vererbung geht das "automatisch"
    MyChild Child;
    MyBase* pBase = NULL;
    pBase = &Child; //impliziter (automat.) statischer Upcast
    pBase = (MyBase*) &Child; //herkömmlicher statischer Upcast
    pBase = static_cast<MyBase*>(&Child); //statischer Upcast
    pBase = dynamic_cast<MyBase*>(&Child); //dynamischer Upcast
    if(!pBase)
        printf("*Upcast* fehlgeschlagen!\n");
    try{ MyBase& rBase = dynamic_cast<MyBase&>(Child); }
    catch(...) { printf("*Upcast* fehlgeschlagen!\n"); }

    //Downcast von Basis-Zeiger (statischer Typ)
    //zu Child-Zeiger/-Referenz (dynamischer bzw.eigentlich.Typ)
    //-> Ok, da in Wirklichkeit auf ein Child gezeigt wird
    pBase = &Child;
    MyChild* pChild = NULL;
    // pChild = pBase; //unmöglich
    pChild = (MyChild*) pBase; //herkömmlicher statischer Downcast
    pChild = static_cast<MyChild*>(pBase); //statischer Downcast
    pChild = dynamic_cast<MyChild*>(pBase); //dyn. Downcast
    if(!pChild)
        printf("*Downcast1* fehlgeschlagen!\n");
    try{ MyChild& rChild = dynamic_cast<MyChild&>(*pBase); }
    catch(...) { printf("*Downcast1* fehlgeschlagen!\n"); }
```

```

//Downcast von Basis-Objekt (dynamischer bzw.eigentlich.Type)
//zu Child-Zeiger (statischer Typ)
//-> Geht nie - funktioniert statisch nur scheinbar
MyBase Base;
pChild = NULL;
// pChild = pBase; //unmöglich
pChild = (MyChild*) &Base; //-> scheinbar gültiger Zeiger!!!
pChild = static_cast<MyChild*>(&Base); //dto.
pChild = dynamic_cast<MyChild*>(&Base); //-> NULL-Zeiger
if(!pChild)
    printf("Downcast2* fehlgeschlagen!\n");

return 0;
}

```

19.4.2 *dynamic_cast* zur Argumentprüfung bei Basisklassen-Zeiger/Referenz

Wenn man eine Funktion schreibt, die als Argument einen Zeiger auf die Basisklasse erwartet oder ein Objekt der Basisklasse referenziert, dann kann man mit `dynamic_cast` **überprüfen, ob der Nutzer ein gültiges Objekt übergeben hat**:

Beispiel:

```

class MyBase
{
public:
    MyBase(int nID = 0) : m_nID(nID) {}
    virtual ~MyBase() {}
    virtual void SetID(int nID);
    virtual int GetID() const;
private:
    int m_nID;
};

void MyBase::SetID(int nID)
{
    m_nID = nID;
}

int MyBase::GetID() const
{
    return m_nID;
}

class MyChild1 : public MyBase
{
public:
    MyChild1(int nID = 0) : MyBase(nID) {}
    virtual ~MyChild1() {}
    void DoAction1()
    {
        printf("Action1\r\n");
    };
};

```

```

class MyChild2 : public MyBase
{
    public:
        MyChild2(int nID = 0) : MyBase(nID) {}
        virtual ~MyChild2() {}
        void DoAction2()
        {
            printf("Action2\r\n");
        };
};

void g(MyBase& r)
{
    try
    {
        MyChild1& rChild1 = dynamic_cast<MyChild1&>(r);
        rChild1.DoAction1();
    }
    catch(...) {}
    try
    {
        MyChild2& rChild2 = dynamic_cast<MyChild2&>(r);
        rChild2.DoAction2();
    }
    catch(...) {}
}

void h(MyBase* p)
{
    MyChild1* pChild1 = dynamic_cast<MyChild1*>(p);
    if(pChild1)
        pChild1->DoAction1();
}

int main()
{
    MyBase Base;
    MyChild1 Child1;
    MyChild2 Child2;

    g(Base); //-> keine Aktion
    h(&Base); //-> keine Aktion
    g(Child1); //-> Action 1 - Child1.DoAction1()
    h(&Child1); //-> Action 1 - Child1.DoAction1()
    h(&Child2); //-> keine Aktion
    g(Child2); //-> Action 2 - Child2.DoAction2()

    return 0;
}

```

19.5 const_cast

Mit `const_cast` kann man `const` beseitigen.

- **Bei Konstanten:**

```
int main()
{
    const int nID = 2;
    (const_cast<int&>(nID)) = 3;
    return 0;
}
```

- **Bei const-Argumenten:**

```
class MyClass
{
public:
    MyClass(int nID = 0) : m_nID(nID) {}
    void SetID(int nID) { m_nID = nID; }
    int GetID() const { return m_nID; }
private:
    int m_nID;
};

void Func(const MyClass& Obj)
{
    int i = Obj.GetID(); //Read-Only-Methoden aufrufbar
    (const_cast<MyClass&>(Obj)).SetID(8);
    i = Obj.GetID();
};

int main()
{
    MyClass Obj(5);
    Func(Obj);
    return 0;
}
```

19.6 reinterpret_cast (!nicht portabel!) und Funktions-Vektoren

Mit `reinterpret_cast` kann man wild zwischen Zeigern auf beliebige Dinge wie Objekte, Funktionen oder Strukturen casten.

Beispiel:

Es sollen Zeiger auf `int`-Funktionen in einen Vektor mit Zeigern auf `void`-Funktionen aufgenommen und wieder ausgelesen werden:

```

typedef void (*pvoidFunc)();
int f1() { return 1; }
int f2() { return 2; }
int f3() { return 3; }
int main()
{
    pvoidFunc arrFuncs[10];
    arrFuncs[0] = reinterpret_cast<pvoidFunc>(f1());
    arrFuncs[1] = reinterpret_cast<pvoidFunc>(f2());
    arrFuncs[2] = reinterpret_cast<pvoidFunc>(f3());
    int i = 0;
    i = reinterpret_cast<int>(arrFuncs[0]);
    i = reinterpret_cast<int>(arrFuncs[1]);
    i = reinterpret_cast<int>(arrFuncs[2]);
    return 0;
}

```

Das `reinterpret_cast` **nicht grundsätzlich portabel** ist liegt auf der Hand, da es hier (ähnlich wie bei der `union`) eine Hardware-Abhängigkeit geben kann, wenn z.B. zwischen **char-Array-Strings** und eingebauten Datentypen gecastet wird. Dies zeigt das folgende Beispiel, welches auf einem **little-endian**-CPU-System einen Fehler liefert, während es auf einem **big-endian**-CPU-System fehlerfrei läuft:

```

int main()
{
    char szValveStates[] = "0010"; //Status der Ventile A,B,C,D
    unsigned long* pdwValveStates
        = reinterpret_cast<unsigned long*>(szValveStates);
    char A = (char) ((*pdwValveStates) >> 24);
    char B = (char) ((*pdwValveStates) >> 16);
    char C = (char) ((*pdwValveStates) >> 8);
    char D = (char) (*pdwValveStates);
    printf("Status der Ventile: ABCD\r\n");
    printf("          ||||\r\n");
    printf("szValveStates:  %s\r\n", szValveStates);
    printf("dwValveStates:   %c%c%c%c", A,B,C,D);
    if(C == '1') //-> funktioniert nicht auf "little-endian"
        printf(" -> Ok\r\n");
    else
        printf(" -> ERROR\r\n");
    return 0;
}

```

19.7 STL: Min- und Max-Werte zu einem Datentyp

Die STL bietet Templates, die einem den minimalen bzw. den maximalen Wert zu einem Datentyp liefern:

```

numeric_limits<Typ>::min()
numeric_limits<Typ>::max()

```

Beispiel:

```

numeric_limits<int>::min()

```

21. Die STL (Standard Template Library)

21.1 Allgemeines

Um nicht Standard-Algorithmen für **Sequenzen** (Listen, Vektoren, ...) von Objekten immer wieder neu implementieren zu müssen (und dies dann noch pro Datentyp einmal) hat man sich etwas einfallen lassen:

- a) Man schreibt einen **Container**, d.h. eine Klasse, die die gesamte Funktionalität für eine Sequenz beinhaltet. Als **sequentiellen Container** bezeichnet man einen solchen, der auf einer Listenstruktur basiert (Bsp.: `list`). **Assoziative Container** bestehen aus 2 Listen: Die Liste der Schlüsselfelder (keys) und die Liste der Wertfelder (values). Wenn man einen Wert (value) hineinschreibt, dann muß man angeben, unter welchem Schlüssel (key) dieser gespeichert werden soll (Bsp.: `map`).
- b) In einem weiteren Schritt sorgt man dafür, dass dieser Container **beliebige Elemente** aufnehmen kann, wozu man das **Template**-Konzept (Vorlagen-Konzept) einführt:

Beispiel für die Definition eines Templates:

```
template<class T>
class SmartPtr
{
public:
    SmartPtr(T* pT = NULL) : m_pT(pT) {}
    SmartPtr(SmartPtr<T>& Obj) : m_pT(Obj.GetPtr())
    {
        Obj.Release();
    }
    ~SmartPtr()
    {
        if(m_pT)
            delete m_pT;
    }
    void Release()
    {
        m_pT = NULL;
    }
    ...
    T* operator->() { return m_pT; }
    T& operator*() { return *m_pT; }
private:
    T* m_pT;
};
```

Instanziierung dieser Vorlage für den Typ `MyClass`:

```
SmartPtr<MyClass> spMyClass(new MyClass);
```

- c) Als letztes schafft man sich dann noch ein **Zugriffsobjekt**, über welches man auf jedes Element der Sequenz zugreifen kann, einen sogenannten **Iterator**:

Beispiel:

```
list<int> listInteger;
listInteger.push_back(105);
listInteger.push_back(99);
listInteger.push_back(308);
listInteger.push_back(66);
list<int>::iterator it;
for(it = listInteger.begin(); it != listInteger.end(); ++it)
{
    if((*it) == 308) //Lesezugriff
        (*it) += 1000; //Schreibzugriff (bei list zulässig)
    printf("%d ", (*it)); //Lesezugriff
}
```

Diese Dinge haben Informatiker schon seit langem erledigt und optimiert, so dass 1994 Alexander **Stepanov** und Meng **Lee** das Ergebnis ihrer langjährigen Forschungsarbeiten im Bereich der Standard-Algorithmen (bei der Firma HP) erfolgreich als **STL (Standard-Template-Library)** in das ISO/ANSI-C++-Werk einbringen konnten.

Es gibt **verschiedene Implementierungen** der STL. Hier ein paar Beispiele für portable Implementierungen:

- STL von STLport <http://www.stlport.org/>
- STL von Rogue Wave http://www.ccd.bnl.gov/bcf/cluster/pgi/pgC++_lib/stdlib.htm
- STL von SGI (Silicon Graphics Inc.) <http://www.sgi.com/tech/stl/>
- STL von HP (Hewlett-Packard) bzw. D.R. Musser <ftp://ftp.cs.rpi.edu/pub/stl/>

Beispiele für Container-Klassen der STL:

vector	→	Eindimensionales Feld
list	→	Doppelt verkettete Liste
queue	→	Schlange
deque	→	Schlange mit 2 Enden
stack	→	Stack
set	→	Sortierte Menge
bitset	→	Sortierte Menge von booleschen Werten
map	→	Sortierte Menge (Schlüsselfelder), die mit anderer Menge (Wertefelder) assoziiert ist

Besonderheiten:

- **Token:**

Die **Token** (Steuerzeichen) < und > sind etwas problematisch: Bei **Verschachtelungen von Templates** kann eine **Verwechslung der doppelten Klammerung mit << oder >>** passieren. Deshalb muss man bei Verschachtelungen ggf. ein **SPACE** einfügen.

Beispiel:

```
Falsch:      set<long, less<long>> setIDs;
Richtig:    set<long, less<long> > setIDs;
                ↑
```

- **Iteratoren:**

Iterator-Objekte arbeiten wie Zeiger und entkoppeln die Algorithmen von den Daten, so dass diese typunabhängig werden. Sie sind praktisch Schnittstellen-Objekte.

Beim **Zugriff auf const-Referenzen** muss man mit

const_iterator

statt `iterator` arbeiten.

- **Effektivität:**

Die STL bietet ein **optimiertes dynamisches Heap-Speicher-Management** mit **generischem Code**, was kaum zu überbieten sein dürfte. So sollte man sich intensiv der STL bedienen, wenn man etwas **"Dynamisches" auf dem Heap** benötigt. Kann man die zu lösende Aufgabe jedoch mittels nicht-dynamischem Array lösen (keine dynamische Länge des Arrays) dann sollte man prüfen, ob die Nutzung eines Arrays auf dem Stack nicht doch effektiver ist.

Beispiel:

```
vector<int> vectValues;
    → Container auf dem Stack, der über die Methode push_back()
       intern Heap allokiert, um Daten zu speichern

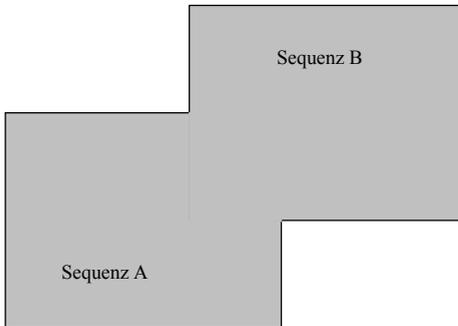
int aValues[100];
    → Array auf dem Stack ("purer" Stack-Speicher)
```

- **Güte der Implementierung und hash_map:**

Möchte man eine Implementierung der STL testen, dann sollte man erst mal einen Blick auf die `hash_map` werfen. Besonders schwach ist die Implementierung, wenn es gar keine `hash_map` gibt. Die Untersuchung der Schnelligkeit der `hash_map` sagt viel aus → wenn so etwas Kompliziertes wie eine `hash_map` gut funktioniert, dann ist das schon mal ein sehr gutes Zeichen. Zum Testen kann man zunächst als Schlüssel (key) den Typ `string` verwenden.

21.5.5 Vereinigungsmenge bilden (*set_union*)

Die STL kann eine Vereinigungsmenge zweier Sequenzen bilden:

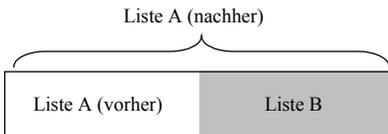


```
listA.sort();
listB.sort();

list<MyClass> listUnion;
set_difference( listA.begin(), listA.end(),
               listB.begin(), listB.end(),
               back_inserter(listUnion));
```

21.5.6 Liste an eine andere Liste anhängen (*list::insert*)

Die STL kann eine Liste an eine andere Liste anhängen:



```
listA.insert(listA.end(), listB.begin(), listB.end());
```

```

//Globale Operatoren:

inline bool operator==(const MyClass& lhs,const MyClass& rhs)
{
    if(lhs.GetID() != rhs.GetID())
        return false;
    return true;
}
inline bool operator!=(const MyClass& lhs,const MyClass& rhs)
{
    if(lhs == rhs)
        return false;
    return true;
}
inline bool operator<(const MyClass& lhs,const MyClass& rhs)
{
    if(lhs.GetID() < rhs.GetID())
        return true;
    return false;
}

```

21.6.3 *Iterator: ++it statt it++ benutzen*

Man sollte den **Präfix-Operator ++it** immer dem Postfix-Operator `it++` vorziehen (Bsp.: im Kopf einer `for`-Schleife), da dessen **Performance besser** ist.

Beispiel:

```

#include <list>
using namespace std;
int main()
{
    list<int> listInts;
    listInts.push_back(2);
    listInts.push_back(1);
    listInts.push_back(1);
    listInts.push_back(3);
    listInts.sort();
    listInts.unique();
    list<int>::iterator it;
    for(it = listInts.begin();it != listInts.end();++it)
        printf("%d\n",(*it));
    return 0;
}

```

Grund:

Der Postfix-Operator benötigt ein **zusätzliches temporäres Objekt**, welches **konstruiert**, für die Rückgabe **kopiert** (Wert-Rückgabe) und dann noch **destruiert** werden muss:

- **Eigene hash-Funktion zur Verfügung stellen:**

```
hash_map<key, value, MyHash> hmapKeyToValue
```

Diese Methode ist auf jeden Fall erforderlich, wenn man eine selbst geschriebene Klasse als key benutzt!

Beispiel:

Der key wird **aus den ersten 4 Zeichen eines Strings** ermittelt. Man verwendet die 8-Bit-ASCII-Codes der Zeichen zum Errechnen eines Indexes:

"ABCD..." → 0xA₁A₀B₁B₀C₁C₀D₁D₀

```
struct MyHash
{
    size_t operator()(const char* pKey) const
    {
        unsigned long dwKey = 0;
        for(int i = 0; i < 4; ++i)
        {
            if(*pKey == 0x00) //end of string
                break;
            dwKey <<= 8;
            dwKey |= (unsigned long) *pKey;
            ++pKey;
        }
        return dwKey;
    }
};

int main()
{
    hash_map<const char*, int, MyHash> hmapMonthToDays;

    hmapMonthToDays["January"] = 31;
    hmapMonthToDays["February"] = 28;
    hmapMonthToDays["March"] = 31;
    hmapMonthToDays["April"] = 30;
    hmapMonthToDays["May"] = 31;
    hmapMonthToDays["June"] = 30;
    hmapMonthToDays["July"] = 31;
    hmapMonthToDays["August"] = 31;
    hmapMonthToDays["September"] = 30;
    hmapMonthToDays["October"] = 31;
    hmapMonthToDays["November"] = 30;
    hmapMonthToDays["December"] = 31;

    printf("September: %d days\n",
           hmapMonthToDays["September"]);

    return 0;
}
```

- **Eigene hash- und equal_to-Funktion zur Verfügung stellen:**

```
hash_map<key, value, MyHash, MyEqualTo> hmapKeyToValue
```

Beispiel:

Der key wird sowohl selbst gehashed (**aus den ersten 4 Zeichen**), als auch auf Gleichheit überprüft.

```
struct MyEqualTo
{
    bool operator()(const char* s1, const char* s2) const
    {
        return strcmp(s1, s2) == 0;
    }
};

struct MyHash
{
    size_t operator()(const char* pKey) const
    {
        unsigned long dwKey = 0;
        for(int i = 0; i < 4; ++i)
        {
            if(*pKey == 0x00) //end of string
                break;
            dwKey <<= 8;
            dwKey |= (unsigned long) *pKey;
            ++pKey;
        }
        return dwKey;
    }
};

int main()
{
    hash_map<const char*, int, MyHash, MyEqualTo>
        hmapMonthToDays;
    hmapMonthToDays["January"] = 31;
    hmapMonthToDays["February"] = 28;
    hmapMonthToDays["March"] = 31;
    hmapMonthToDays["April"] = 30;
    hmapMonthToDays["May"] = 31;
    hmapMonthToDays["June"] = 30;
    hmapMonthToDays["July"] = 31;
    hmapMonthToDays["August"] = 31;
    hmapMonthToDays["September"] = 30;
    hmapMonthToDays["October"] = 31;
    hmapMonthToDays["November"] = 30;
    hmapMonthToDays["December"] = 31;

    printf("September: %d days\n",
        hmapMonthToDays["September"]);

    return 0;
}
```

22. Arten von Templates

22.1 Class-Template

Vorlage für eine Klasse. Hier gezeigt am Beispiel einer Smart-Pointer-Klasse:

```
template<class T>
class SmartPtr
{
public:
    SmartPtr(T* pT = NULL) : m_pT(pT) {}
    SmartPtr(SmartPtr<T>& Obj)
        : m_pT(Obj.GetPtr()) //Zeiger vom Eigentümer nehmen
    {
        Obj.Release(); //Obj von Eigentümerschaft befreien
    }
    ~SmartPtr()
    {
        if(m_pT)
            delete m_pT;
    }
    void Release()
    {
        m_pT = NULL; //von Eigentümersch.befreien
    }
    T* GetPtr() { return m_pT; }
    bool IsValid()
    {
        if(m_pT != NULL)
            return true;
        return false;
    }
    operator bool() { return IsValid(); }
    SmartPtr<T>& operator=(SmartPtr<T>& Obj)
    {
        if(this == &Obj)
            return *this;
        if(m_pT)
            delete m_pT; //alten Speicher freigeben
        m_pT = Obj.GetPtr(); //Eigentümerschaft übernehmen
        Obj.Release(); //Obj von Eigentümersch.befreien
        return *this;
    }
    T* operator->() { return m_pT; }
    T& operator*() { return *m_pT; }
private:
    T* m_pT;
};
```

Ein Objekt dieses Templates wird folgendermaßen instanziiert, wenn man den Typ `T = MyClass` benutzt:

```
SmartPtr<MyClass> spMyClass(new MyClass);
```

22.2 Function-Template

22.2.1 Global Function Template

Vorlage für eine globale Funktion. Hier gezeigt am Beispiel der Maximum-Ermittlung:

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

Eine solche Funktion wird ganz normal aufgerufen:

```
int i = MyMax(3,4);           //→ i == 4
char c = MyMax('k','l');     //→ c == 'l'
double d = MyMax(2.5,4.6);   //→ d == 4.6
```

Der Compiler erkennt den Typ T und setzt ihn richtig ein.

22.2.2 Member Function Template

Vorlage für eine Member-Funktion.

Beispiel:

```
#include <typeinfo>
class MyClass1
{
    public:
        MyClass1() {}
        ~MyClass1() {}
};
class MyClass2
{
    public:
        MyClass2() {}
        ~MyClass2() {}
};

class Helper
{
    public:
        Helper() {}
        ~Helper() {}
        template<class T, class C>
        bool IsEqualType(const T& lhsObj, const C& rhsObj);
};
```

25. Aktion nach Kollision über Objekttyp steuern

Problem:

Eine Funktion (Handler) soll das Aufeinandertreffen von 2 Objekten handeln. Die auszuführende Aktion soll dabei davon abhängen, welche Objekt-Typen aufeinandertreffen.

Lösung:

Zunächst schafft man sich eine Handler-Map, die alle möglichen Handler aufnimmt:

```
class Base; //Forward-Deklaration

typedef void (*HitFuncPtr) (Base&,Base&);

class HandlerMap
{
public:
    static void AddEntry( unsigned short wID1,
                        unsigned short wID2,
                        HitFuncPtr Handler);
    static HitFuncPtr Lookup( unsigned short wID1,
                            unsigned short wID2);
private:
    HandlerMap(); //verstecken
    HandlerMap(const HandlerMap&); //verstecken
private:
    static map<unsigned long,HitFuncPtr>&
                                                theHandlerMap();
};
```

In diese Map kann man für jede mögliche Kombination nID1 mit nID2 einen Handler, z.B.

```
void Handler13(Base& Obj1,Base& Obj3)
{
    ...
}
```

aufnehmen:

```
HandlerMap::AddEntry(1,3,&Handler13);
```

Die Klassen aller Objekte werden von folgender abstrakter Basisklasse abgeleitet:

```
class Base
{
public:
    virtual ~Base() {}
    virtual void Handler(Base& Partner);
    virtual unsigned short GetID() const = 0;
};
```

```

void Base::Handler(Base& Partner)
{
    HitFuncPtr hfHandler
        = HandlerMap::Lookup(GetID(), Partner.GetID());
    if(hfHandler != NULL)
        hfHandler(*this, Partner);
}

```

Hier ein Beispiel:

```

class Child1 : public Base
{
    public:
        virtual unsigned short GetID() const;
};
unsigned short Child1::GetID() const
{
    return 1;
}

```

Eine **Kollision** sieht wie folgt aus:

```

Child1 Obj1;
Child2 Obj2;
Child1.Handler(Child2); //1 kollidiert mit 2

```

Es passieren also 2 Dispatches (**Double Dispatching**):

- Erster Dispatch:

Der Kollisions-Partner Obj2 wird an den Handler des kollidierenden Objektes Obj1 weitergeleitet.

- Zweiter Dispatch:

Der Handler des kollidierenden Objektes Obj1 sucht in der Handler-Map den Kollisions-Handler für die Paarung Obj1 mit Obj2 und gibt sich selbst und den Partner an den Handler weiter.

Und hier nun das komplette Beispiel 'am Stück':

```

//----- STL: -----
#include <stdio.h>
#include <map>
using namespace std;

```

```

//----- HandlerMap: -----
class Base; //Forward-Deklaration
typedef void (*HitFuncPtr) (Base&,Base&);

class HandlerMap
{
public:
    static void AddEntry(unsigned short wID1,
                        unsigned short wID2,
                        HitFuncPtr Handler);
    static HitFuncPtr Lookup( unsigned short wID1,
                            unsigned short wID2);
private:
    HandlerMap(); //verstecken
    HandlerMap(const HandlerMap&); //verstecken
private:
    static map<unsigned long,HitFuncPtr>& theHandlerMap();
};

map<unsigned long,HitFuncPtr>& HandlerMap::theHandlerMap()
{
    static map<unsigned long,HitFuncPtr> map_;
    return map_;
}

void HandlerMap::AddEntry(
    unsigned short wID1,unsigned short wID2,HitFuncPtr Handler)
{
    unsigned long dwID = (wID1 << 16) + wID2;
    theHandlerMap()[dwID] = Handler;
}

HitFuncPtr HandlerMap::Lookup(
                                unsigned short wID1,unsigned short wID2)
{
    unsigned long dwID = (wID1 << 16) + wID2;
    map<unsigned long,HitFuncPtr>::iterator it
        = theHandlerMap().find(dwID);
    if(it != theHandlerMap().end())
        return (*it).second;
    return NULL;
}

```

```

//----- Kollidierende Objekte: -----

class Base
{
    public:
        virtual ~Base() {}
        virtual void Handler(Base& Partner);
        virtual unsigned short GetID() const = 0;
};

void Base::Handler(Base& Partner)
{
    HitFuncPtr hfHandler
        = HandlerMap::Lookup(GetID(), Partner.GetID());
    if (hfHandler != NULL)
        hfHandler(*this, Partner);
}

class Child1 : public Base
{
    public:
        virtual unsigned short GetID() const;
};

unsigned short Child1::GetID() const
{
    return 1;
}

class Child2 : public Base
{
    public:
        virtual unsigned short GetID() const;
};

unsigned short Child2::GetID() const
{
    return 2;
}

class Child3 : public Base
{
    public:
        virtual unsigned short GetID() const;
};

unsigned short Child3::GetID() const
{
    return 3;
}

```

```
//----- Handler: -----

void Print(unsigned short wIDA,unsigned short wIDB)
{ printf("%d kollidiert mit %d\n",wIDA,wIDB); }

void Handler11(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler12(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler13(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler21(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler22(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler23(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler31(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler32(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }
void Handler33(Base& ObjA,Base& ObjB)
{ Print(ObjA.GetID(),ObjB.GetID()); }

//----- main(): -----

int main()
{
    HandlerMap::AddEntry(1,1,&Handler11);
    HandlerMap::AddEntry(1,2,&Handler12);
    HandlerMap::AddEntry(1,3,&Handler13);
    HandlerMap::AddEntry(2,1,&Handler21);
    HandlerMap::AddEntry(2,2,&Handler22);
    HandlerMap::AddEntry(2,3,&Handler23);
    HandlerMap::AddEntry(3,1,&Handler31);
    HandlerMap::AddEntry(3,2,&Handler32);
    HandlerMap::AddEntry(3,3,&Handler33);

    Child1 Obj1;
    Child2 Obj2;
    Child3 Obj3;

    Obj1.Handler(Obj2);
    Obj2.Handler(Obj1);
    Obj3.Handler(Obj1);

    return 0;
}

```

```

int main()
{
    MyClass Obj1(1);
    MyClass Obj2(2);
    list<MyClass> listObjs;
    listObjs.push_back(Obj1);
    listObjs.push_back(Obj2);
    listObjs.push_back(Obj1);
    list<MyClass>::iterator it
        = find(listObjs.begin(), listObjs.end(), Obj1);
    if(it != listObjs.end())
    {
        int nDeletedID = (*it).GetID()
        listObjs.erase(it); //nicht listObjs.erase(*it)!!!
    }
    return 0;
}

```

26.2.8 map: nie indizierten Zugriff [] nach find() durchführen

Der **indizierte Zugriff** auf eine map (operator[]) erfordert zunächst immer ein **internes find()** zum Suchen des key, um aus dessen Position auf die Position des value zu schließen. Wenn man gerade erst find() ausgeführt hat, um zu wissen, ob sich der key in der Sequenz befindet, dann verschwendet man nochmal die Zeit für ein internes find().

Beispiel:

```

class MyClass
{
public:
    explicit MyClass(int nID = 0) : m_nID(nID) {}
    int GetID() const
    {
        return m_nID;
    }
    bool operator==(const MyClass& Obj) const
    {
        if(Obj.GetID() == m_nID)
            return true;
        return false;
    }
    bool operator<(const MyClass& Obj) const
    {
        if(m_nID < Obj.GetID())
            return true;
        return false;
    }
private:
    int m_nID;
};

```

```

#include <stdio.h>
#include <algorithm>
#include <map>
using namespace std;

int main()
{
    MyClass Obj1(1);
    MyClass Obj2(2);
    map<long,MyClass> mapHandleToObj;
    mapHandleToObj[10] = Obj1;
    mapHandleToObj[300] = Obj2;
    map<long,MyClass>::iterator it = mapHandleToObj.find(10);
    if(it != mapHandleToObj.end())
    {
        MyClass DeletedObj = (*it).second;
        //nicht DeletedObj = mapHandleToObj[10]!!!
        mapHandleToObj.erase(it);
        //nicht mapHandleToObj.erase(*it)!!!
    }
    return 0;
}

```

26.2.9 Unsichtbare temporäre Objekte vermeiden

In folgenden Fällen werden unsichtbare temporäre Objekte erzeugt, d.h. es findet eine unsichtbare Konstruktion und eine Destruktion statt:

- **Übergabe eines Funktions-Argumentes per Wert**

→ Es wird mit einem temporären Objekt statt mit dem Originalparameter gearbeitet.

Abhilfe:

Parameter per const-Referenz übergeben!
--

Beispiel:

Statt:

```
Func(list<long> listIDs);
```

Besser:

```
Func(const list<long>& listIDs);
```

- **Objekte als return-Wert einer Funktion**

→ Es wird ein temporäres Objekt für die Rückgabe erzeugt.

Abhilfe:

Rückgabe eines **Konstruktors** statt eines Objektes!

Die "**return-value-optimization**" des Compilers kann den Konstruktoraufruf direkt an das Objekt des Aufrufers weitergeben, ohne ein temporäres Objekt zu erzeugen.

Beispiel:

Statt:

```
MyClass Obj(14);  
return Obj;
```

Besser:

```
return MyClass(14);
```

- **Übergabe eines Funktions-Argumentes vom 'falschen' Typ**

→ Es wird über eine **implizite Typumwandlung** ein temporäres Objekt erzeugt und übergeben.

Beispiel:

```
int Func(const MyClass2& Obj)  
{  
    int i = Obj.GetID();  
    return i;  
}  
  
int main()  
{  
    MyClass1 Obj(1);  
    Func(Obj);  
    return 0;  
}
```

Beispiel für die Implementierung einer Klasse mit Object-Pooling:

```
#include <new.h>
#include <list>
using namespace std;

class MyClass
{
public:
    MyClass() {}
    void Destroy() { delete this; }
    static void* operator new(size_t size);
    static void operator delete(void* pMem);
private:
    ~MyClass() {}
    static void* operator new[](size_t size); //versteckt
    static void operator delete[](void* pMem); //versteckt
    enum { POOL_SIZE = 1024*1024 }; //1 MB
    static void* Pool(bool bNew);
    static list<MyClass*>& ReleasedItems();
    static unsigned long& NumConstructedItems();
};

void* MyClass::operator new(size_t size)
{
    if(size != sizeof(MyClass))
        return ::operator new(size);

    void* pPool = Pool(true);

    static MyClass* pMem = NULL;
    if(!NumConstructedItems())
        pMem = (MyClass*) pPool;

    MyClass* pItem = NULL;
    if(ReleasedItems().empty())
    {
        pItem = ::new (pMem) MyClass; //Placement-new
        pMem += sizeof(MyClass);
        ++(NumConstructedItems());
    }
    else
    {
        pItem = ReleasedItems().front();
        ReleasedItems().pop_front();
    }

    return pItem;
}
```

```

void MyClass::operator delete(void* pMem)
{
    if(pMem == NULL)
        return;

    MyClass* pItem = (MyClass*) pMem;
    ReleasedItems().push_back(pItem);
    if(ReleasedItems().size() == NumConstructedItems())
    {
        Pool(false);
        ReleasedItems().clear();
        NumConstructedItems() = 0;
    }
}

void* MyClass::Pool(bool bNew)
{
    static void* pPool = NULL;

    if(bNew)
    {
        if(pPool == NULL)
            pPool = operator new(POOL_SIZE);
    }
    else
    {
        if(pPool != NULL)
        {
            delete pPool;
            pPool = NULL;
        }
    }
    return pPool;
}

list<MyClass*>& MyClass::ReleasedItems()
{
    static list<MyClass*> list_;
    return list_;
}

unsigned long& MyClass::NumConstructedItems()
{
    static unsigned long dw = 0;
    return dw;
}

int main() //pA, pB, pC und pD sind zu beobachten
{
    MyClass* pA = new MyClass; //Pool allokiert -> Obj1 platziert
    MyClass* pB = new MyClass; //Obj2 platziert
    pA->Destroy(); //Obj1 gepooled
    MyClass* pC = new MyClass; //Obj1 wieder referenziert
    pB->Destroy(); //Obj2 gepooled
    pC->Destroy(); //Obj1 gepooled -> Pool gelöscht
    MyClass* pD = new MyClass; //Pool allokiert -> Obj1' platziert
    pD->Destroy(); //Obj1' gepooled -> Pool gelöscht
    return 0;
}

```