

Objektorientierte Programmierung (Programmieren 2)

(Teile 02 - 17, kompakt)

mit C++

Prof. Dr. Bernd Ruhland

Warum C++ ?

- C++ ist die Programmiersprache, mit der sich die Objektorientierte Programmierung „in der Breite“ etabliert hat (dank Bjarne Stroustrup)
- C++ hat die Objekt-Kapsel, die Abstraktion zu Klassen, die Vererbung, die Methodenüberladung, den Polymorphismus und andere Grundprinzipien der Objektorientierung
- C++ ist die „Mutter“ von Java (Sun Microsystems) und C# (Microsoft)

Warum Microsoft Visual Studio ?

- MS Windows ist das am weitesten verbreitete Desktop-Betriebssystem
- MS Visual Studio ist das auf MS Windows ausgerichtete IDE (Integrated Development Environment)
- MS Visual Studio bildet den Entwicklungsprozess vollständig ab
- Sie können am Beispiel MS Visual Studio den Umgang mit IDEs generell verstehen und erlernen
- MS Visual Studio unterstützt viele andere Programmiersprachen und deren Laufzeitumgebungen

Warum unter Microsoft Windows ?

- MS Visual Studio ist auf MS Windows ausgerichtet, nur dort erleben Sie es „nativ“
- MS Windows passt sich dem MS Visual Studio bei dessen Installation an (Debugging/Tracing der Kernel-Routinen)
- MS Windows sollte man kennen, auch wenn man es selbst nicht täglich verwendet
- MS Windows hat eine spannende Geschichte:
 - 1985 Windows 1.0 auf Basis von MS DOS (16 Bit) und mit „Anregungen“ vom Apple Macintosh
 - 1992 Windows 3.1 erstmals „spieltauglich“, dann 1996 Windows 95A und 1999 Windows 98SE (erstmals „geschäftstauglich“), dann 1999 das unselige Windows ME: innen Windows 98, außen Windows 2000)
 - 1996 Windows NT (erstmals mit Microkernel, virtuelle Adressräume), dann 2000 Windows 2000, 32 Bit und ein „echtes“ Betriebssystem
 - Windows XP, Windows 7, Windows 8, Windows 10 kennen Sie

1. Vorlesung

Die Programmiersprache C++

- C++ ist, bis auf wenige Ausnahmen, eine Obermenge von C. Es sind alle Bibliotheken, auch die Standardbibliothek, uneingeschränkt verwendbar.
- Die grundlegende Spracherweiterung, die C++ gegenüber C bietet, ist die Möglichkeit, *Klassen* zu definieren.
- Andere neue Sprachelemente stehen in engem Zusammenhang mit dem Klassenkonzept. Einige Neuerungen sind jedoch auch ohne Verwendung von *Klassen* sinnvoll einsetzbar.

Klassen

- Unter *Kapselung* versteht man die Verbindung von Datenelementen und Verarbeitungsschritten (Funktionalitäten) zu einer Einheit (Objekte, abstrahiert zu Klassen).
- Eine Klasse soll alle Daten (Attribute) und Funktionen (Methoden) enthalten, um ein definiertes, genau abgegrenztes Problem zu lösen. Daten und Funktionen einer Klasse sind zunächst privat (*private*).
- Um mit der Außenwelt zu kommunizieren, werden spezielle Methoden verwendet, die in der Klasse explizit als öffentlich (*public*) definiert werden müssen.
- Klassen können zur Definition von Objekten, also von „Variablen“ herangezogen werden. Diese Variablen sind *Instanzen* der Klassen.

Code-Beispiel Klassen

- Klassendefinition:

```
class complexT
{
    double re, im;
};
```

- Definition einer Variablen sowie Generierung einer Instanz (Instanzierungen von Objekten der Klasse):

```
int main() {
    complexT obj1; // Variablendefinition
    complexT *cp1 = new complexT; // Generierung
                                // ... Anweisungen ...
    delete cp1; // Aufloesung
}
```


Klassen in C++

- *Klassen* können als Erweiterung der C-*structs* aufgefasst werden. Der Unterschied besteht darin, dass eine *Klasse* neben Datenelementen (Variablen) auch Verarbeitungselemente (Methoden) enthalten kann.

Beispiel:

```
class cFahrzeug {  
    private:  
        double laenge, hoehe, breite;  
        int herstelljahr;  
    public:  
        void bewegen();  
};
```

- Der Programmentwickler bestimmt die Sichtbarkeit jedes einzelnen Elements einer *Klasse*.

Kurzeinstieg Streams

Streams

(vormals Teil06)

- In C stehen zur Ein- Ausgabe von Daten Funktionen wie *printf* und *scanf* zur Verfügung. C++ beinhaltet zusätzlich so genannte *Streams*.

```
//    Hello World mit Streams
#include <iostream>
using namespace std;    // Erweiterung Namensraum

int main() {

    cout << "hello world .... ";

    return 0;
}
```

- *cout* ist der Standard-Ausgabestrom, vergleichbar mit *stdout* in C

Streams

//-- Eingabe eines Integers mit Streams

```
#include <iostream>
```

```
using namespace std;
```

```
int main()  {
```

```
    cout << "Bitte einen Wert für i eingeben : ";
```

```
    int i;
```

```
    cin >> i;
```

```
    cout << "Der eingegebene Wert ist: " << i << endl;
```

```
    return 0;
```

```
}
```

**Ein-/Ausgabe
beide Richtungen**

- *cin* ist der Standard-Eingabestrom, der normalerweise mit der Tastatur verbunden ist (stdin).

Ausgabe unterschiedlicher Datentypen

- Die Operatoren << und >> sind überladen für alle einfachen Datentypen und führen die eigentliche Ausgabe bzw. Eingabe für die unterschiedlichen Datentypen aus.

//-- Ausgabe unterschiedlicher Datentypen mit <<

```
#include <iostream>
using namespace std;
```

```
int main () {
    char* s = "Ein String";
    int    i = 1;
    float  f = 3.1415;
    void*  p = &i;
```

**automatisch zum
Datentyp passend**

```
    cout << '\n'; cout << "string      :"; cout << s;
    cout << '\n'; cout << "integer   :"; cout << i;
    cout << '\n'; cout << "float     :"; cout << f;
    cout << '\n'; cout << "Zeiger    :"; cout << p;
}
```

Kaskadierung

- Dadurch können die Operatoren kaskadiert werden:

```
int main () {  
  
    char* s = "Ein String";  
    int i = 1;  
    float f = 3.1415;  
    void* p = &i;  
  
    cout << '\n' << "string" << " " << s  
         << '\n' << "integer" << " " << i  
         << '\n' << "float" << " " << f  
         << '\n' << "Zeiger" << " " << p;  
  
}
```

**verkettbar zu
einer Anweisung**

- Die Priorität eines Operators wird beim Überladen nicht geändert.
- In Kaskaden werden die Übergabeoperatoren von links nach rechts ausgewertet.

Formatierung durch Manipulatoren

- Weitere Manipulatoren:

endl	Neue Zeile beginnen
ends	Stringendezeichen anhängen
flush	Ausgabepuffer leeren

gekapselte Sonderfunktionen

```
//-- Verwendung des Manipulators endl
```

```
//-- Ausgabe eines Betrags mit Hilfe von Füll-
//      zeichen in einem Feld der Breite 10
```

```
int amount = 355;
```

```
cout << endl << "Sie erhalten heute EUR "  
        << amount << endl;
```

Ende Kurzeinstieg Streams

Übungsaufgabe:

- a) Erstellen Sie ein Hello-World-Programm in C++
- b) Lesen Sie mittels cin Werte ein in Variablen von den Typen:
 - a) Integer
 - b) Double
 - c) Zeichenkette
 - d) Einzelnes Zeichen (schwierig! zum Schluss machen!)
- c) Geben Sie die Werte mittels cout wieder aus

Konstruktoren

- *Konstruktoren* übernehmen die Aufgabe der Initialisierung der Datenelemente der *Klasse*. Sie haben die „Verantwortung“ dafür, dass sich ein Objekt bereits zum Zeitpunkt seiner Erzeugung in einem definierten Zustand befindet. Hierunter fällt auch die Anforderung von dynamischem Speicher, falls das *Objekt* mit Daten auf dem *Heap* arbeitet.
- Der *Konstruktor* wird bei der Definition eines *Objekts* automatisch aufgerufen und kann mit Parametern ausgestattet sein, die zur Initialisierung verwendet werden können.

Codebeispiel Konstruktor

```
class cFahrzeug {           // Klassendefinition
private:
    double laenge, hoehe, breite;  int herstelljahr;
public:
    cFahrzeug() {           // Standard-Konstruktor
        laenge=0.0; hoehe=0.0; breite=0.0; herstelljahr=0;
    }

    // Konstruktor mit Parametern
    cFahrzeug(double l, double h, double b, int j) {
        laenge=l; hoehe=h; breite=b; herstelljahr=j;
    }
};

int main() {
    // Parameterübergabe bei der Objektdefinition
    cFahrzeug UtesFlitzer(3.85, 1.20, 1.60, 2006);
    // ...
}
```

Vorgabewerte

- *Vorgabewerte für Argumente* : Bei der Definition einer C++ Funktion können den formalen Parametern Vorgabewerte zugeordnet werden. Werden beim Aufruf der Funktion aktuelle Parameter in der Parameterliste weggelassen, erhalten die fehlenden Parameter die Vorgabewerte.
- *Konstruktoren* können Vorgabeparameter (Parameter mit Vorgabewerten, „Standardparameter“) enthalten.
- Parameter mit Vorgabewerten müssen rechts von Parametern ohne Vorgabewerte stehen.
- Sind alle Parameter mit Vorgabewerten ausgestattet, ist der Konstruktor ein „universeller Konstruktor“, er ersetzt dann auch den Standardkonstruktor.

Destruktoren

- *Destruktoren* übernehmen Restarbeiten, die dann erforderlich werden, wenn das Objekt nicht mehr benötigt wird (z. B. Freigabe von dynamisch angefordertem Speicher).
- Der *Destruktor* einer *Klasse* wird automatisch aufgerufen, wenn der Gültigkeitsbereich eines *Objekts* verlassen wird.
- Der Destruktor hat den Namen der zugehörigen Klasse mit einem vorangestellten Tilde-Zeichen, '~'.

Beispiel:

```
cFahrzeug::~~cFahrzeug() { // KEINE Parameter
    cout << „Objekt Fahrzeug wird zerstört“ << endl;
}
```

Kurzeinstieg: strings

Der Datentyp string

- ist eine Klasse
(wie auch alle elementaren Datentypen in C++)
- besitzt somit Konstruktoren, die aus Zeichenketten (Variablen oder Konstanten) einen String aufbauen
- versorgt sich selbst mit dem nötigen Speicherplatz, auch bei Änderungen an der Stringlänge
- besitzt einen Destruktor, der den Speicherplatz wieder freigibt
- hat Eigenschaften, die abgefragt werden können
- besitzt dazu Schnittstellenfunktionen

strings

...

- hat Mitgliedsfunktionen für die Basis-Stringoperationen
- hat weitere Mitgliedsfunktionen für komplexere Stringoperationen
- unterstützt die meisten Operatoren, z.B.
 - Zuweisungen
 - Vergleiche
 - „Rechen“-Operationen
 - ...

➔ siehe z.B. Intellisense des MS VisualStudio

Codebeispiel strings (Ende Kurzeinstieg)

```
#include <iostream>
#include <string>          // notwendig !
using namespace std;

int main () {
    string mystr, mystr2;

    mystr = "MyString wird besetzt";

    cout << "Ausgabe mystr: " << mystr << endl;

    mystr.append(" und verlaengert");
    cout << "Ausgabe mystr: " << mystr << endl;

    mystr2 = mystr.substr(13);
    cout << "Ausgabe mystr2: " << mystr2 << endl;

    return 0;
}
```

2.Vorlesung

Operatoren *new* und *delete*

- *Konstruktor*en bzw. *Destruktor*en werden bei der Verwendung von *new* und *delete* automatisch aufgerufen.
- *new* und *delete* sind keine Funktionen sondern Operatoren.

Anforderung dynamischen Speichers

```
Fahrzeug * p; // Definition eines Zeigers
```

```
p = (Fahrzeug*) malloc(sizeof(Fahrzeug)); // in C
```

```
p = new Fahrzeug; // in C++
```

- Eine explizite Angabe der Größe des zu reservierenden Speicherbereiches sowie die Typumwandlung ist bei der Verwendung von *new* nicht mehr erforderlich.

Zugriffsoperatoren

Beispielkasse:

```
class complexT
{
    private:
        double re, im;

    public:
        void set(double re_in, double im_in) {
            re = re_in;
            im = im_in;
        }

        void print() {
            cout << "re = " << re << ", im = " << im << endl;
        }
};
```

Zugriffsoperatoren

Funktionen außerhalb der Klasse können auf Klassenmitglieder nur über einen der Zugriffsoperatoren "." oder "->" zugreifen.

```
int main ()
{
    complexT x1, *xp;

    x1.set (1.0, 2.0);    x1.print ();

    xp = new complexT;

    xp->set (3.0, 4.0);    xp->print ();
    // Kurzschreibweise fuer:
    // (*xp).set (3.0, 4.0); (*xp).print ();

    delete xp;
}
```

Schlüsselwort *this*

- Das Schlüsselwort *this* ermöglicht den Zugriff auf ein Objekt um dessen Adresse in Erfahrung zu bringen.
- Der *this*-Zeiger wird vom Compiler automatisch deklariert und zur Laufzeit mit der Adresse des Objektes besetzt (konstanter Zeiger).

Datenelemente von Klassen

- Die Datenelemente der Klasse werden genau wie die Elemente einer Struktur angegeben, jedoch ist eine Initialisierung der Datenelemente nicht erlaubt.
- Datenelemente einer Klasse können nicht als const deklariert werden.
- Datenelemente können von beliebigem Typ sein. Insbesondere ist es möglich, als Typ auch Felder, Strukturen oder andere Klassen anzugeben.

Codebeispiel Aggregation

```
// Class OneRealT
```

```
class OneRealT
{
private:
    double r;
public:
    void set (double r_in){ r = r_in; };
    void print (void) { cout << "r = " << r << endl; };
};
```

```
// Neue Klassendefinition für complexT
```

```
class complexT
{
private:
    OneRealT re, im;      // Objekte der Klasse s.o.
public:
    void set (double re_in, double im_in);
    void print (void);
};

void complexT::set ( double re_in, double im_in)
{
    re.set (re_in);
    im.set (im_in);
}
```

3. Vorlesung

Gültigkeitsbereich und Existenz

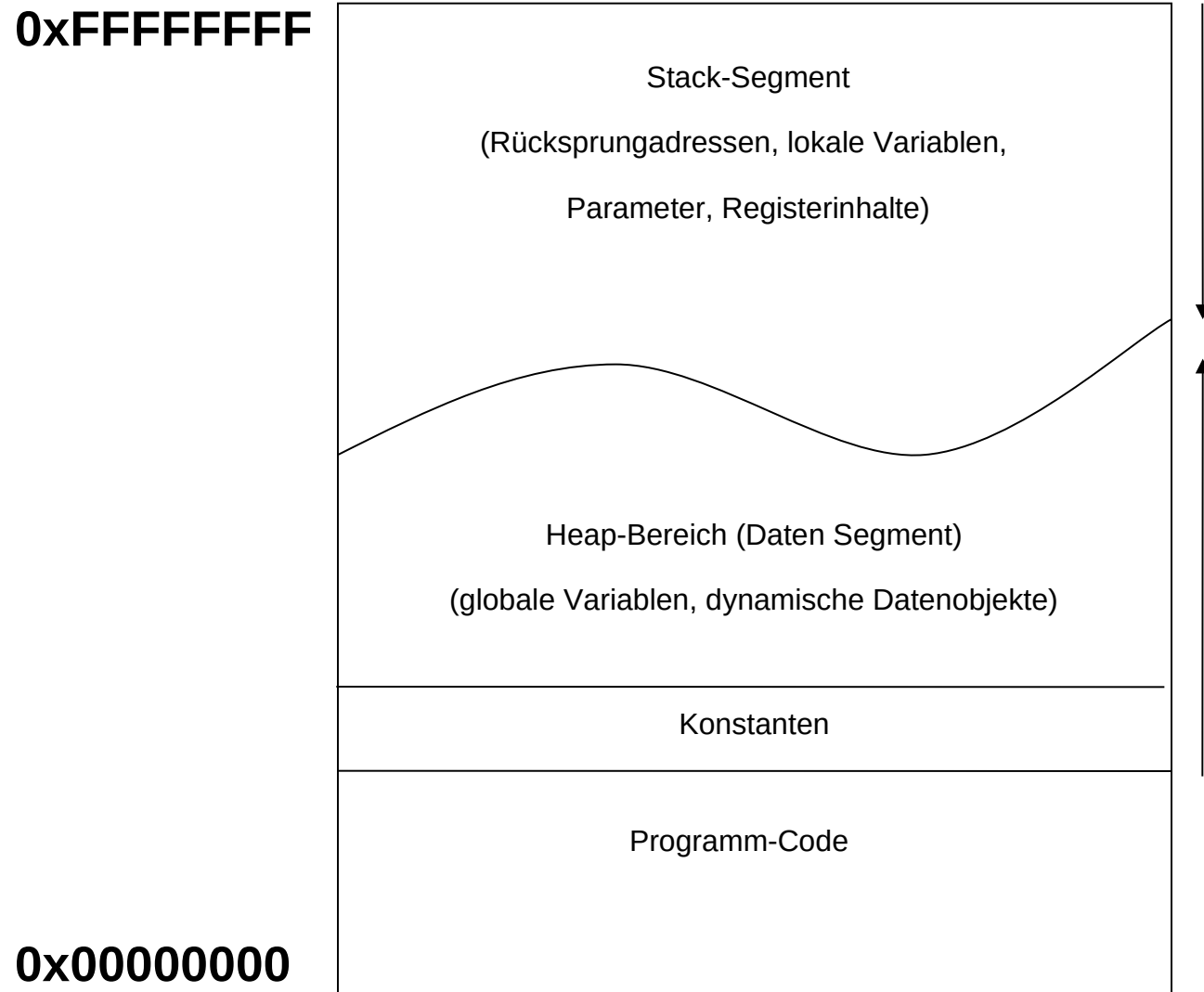
- Der Datenbereich für lokale Daten wird bei Betreten des Gültigkeitsbereichs auf einem besonderen Speicherbereich mit dem Namen *Stack* angelegt und am Ende des Gültigkeitsbereichs, also am Blockende, wieder freigegeben.
- Der Stack (deutsch: »Stapel«, auch Kellerspeicher) ist ein Bereich mit der Eigenschaft, dass die zuletzt darauf abgelegten Elemente zuerst wieder freigegeben werden (last in, first out).
- Der Stack gehört zum Daten-Segment des Prozesses im Hauptspeicher, bei manchen Betriebssystemen ist er sogar ein eigenes Speichersegment.

Was ist mit dynamischen Variablen? (durch new erzeugt)

Gültigkeitsbereich und Existenz

- Der Datenbereich für durch *new* erzeugte „dynamische“ Objekte besonderen Speicherbereich mit dem Namen *Heap* angelegt und erst durch den Operator *delete* wieder freigegeben. (Fehlt das *delete*, wird das Objekt erst beim Programmende aufgelöst!)
- Der *Heap* gehört zum Daten-Segment des Prozesses im Hauptspeicher, bei manchen Betriebssystemen ist er sogar ein eigenes Speichersegment.

Speicherverwendung



Gültigkeitsbereich und Sichtbarkeit

In C++ gelten Gültigkeits- und Sichtbarkeitsregeln für Namen. Es gibt folgende Regeln (bekannt aus C):

- Namen sind nur *nach der Deklaration* und nur *innerhalb des Blocks* gültig, in dem sie deklariert wurden. Sie sind *lokal* bezüglich des Blocks. Zur Erinnerung: Ein Block ist ein Programmbereich, der durch ein Paar geschweiffter Klammern { } eingeschlossen wird. Blöcke können verschachtelt sein, also selbst wieder Blöcke enthalten.
- Namen von Variablen sind auch gültig für innerhalb des Blocks neu angelegte innere Blöcke.
- Klassenrümpfe (Körper) sind ebenfalls Blöcke in diesem Sinn. Nur das, was wir mit dem Zugriffsrecht „public“ definieren, ist außerhalb der Klasse sichtbar.

Gültigkeitsbereich und Sichtbarkeit

Neu in C++: **Namensräume** (namespaces):

- Benannter Block auf globalem Niveau:

```
namespace boerse {  
    const int daxWerte = 31;  
    double aktienKurs (int WertPapierNummer);  
}
```

- Zugriff auf namespace mittels using:

```
using boerse::aktienKurs; // nur einzelne Funktion
```

```
using namespace boerse;    // gesamten Namensraum
```

- Die Schachtelung von Namensräumen ist möglich

Gültigkeitsbereich und Sichtbarkeit

- Die Sichtbarkeit (englisch *visibility*) zum Beispiel von Variablen wird eingeschränkt durch die Deklaration von Variablen gleichen Namens. Für den Sichtbarkeitsbereich der inneren Variablen ist die äußere unsichtbar.
- In C++ kann die Sichtbarkeit verdeckter Variablen oder Funktionen mit dem **Bereichsoperator ::** erzwungen werden.

Beispiel (In C-Schreibweise mit C++-Erweiterung!):

```
int i=1;
int main() {
    int i=2;
    cout << "i innen: " << i << endl;
    cout << "i aussen: " << ::i << endl;
}
```

Codebeispiel II Konstruktor

Prog2_Konstruktor_Flitzer.cpp

```
class cFahrzeug {           // Klassendefinition
private:
    double laenge, hoehe, breite;  int herstelljahr;
public:
    cFahrzeug();
    cFahrzeug(double l, double h, double b, int j);
};

cFahrzeug::cFahrzeug() {           // Standard-Konstruktor
    laenge=0.0; hoehe=0.0; breite=0.0; herstelljahr=0; }

// Konstruktor mit Parametern
cFahrzeug::cFahrzeug(double l, double h, double b, int j) {
    laenge=l; hoehe=h; breite=b; herstelljahr=j; }

int main() {
    // Parameterübergabe bei der Objektdefinition
    cFahrzeug UtesFlitzer(3.85, 1.20, 1.60, 2006);
    // ...
}
```

Mitgliedsfunktionen („lokale Methoden“)

Mitgliedsfunktionen (Methoden) von Klassen unterscheiden sich von gewöhnlichen C-Funktionen in folgenden Punkten:

- Der Gültigkeitsbereich einer Methode ist auf die Klasse beschränkt (class scope).
- Eine Methode kann automatisch auf alle Datenelemente der Klasse (Attribute) zugreifen, ohne diese deklarieren zu müssen oder als Parameter zu erhalten.

Vererbung

(vormals Teile 12-14)

- Einmal vorhandene Klassen können zur Definition neuer Klassen verwendet werden.
- Die Klasse, die zur Definition verwendet wird, heißt *Basisklasse*, die neue Klasse wird *abgeleitete Klasse* genannt.

Vererbung

- Unter *Vererbung* versteht man die Tatsache, dass in einer Klassenhierarchie die abgeleiteten Klassen automatisch die Daten und Funktionen der zugrundeliegenden Klassen (Basisklassen) besitzen.
- In einer abgeleiteten Klasse können zusätzliche Daten und Funktionen definiert sowie geerbte Prozeduren umdefiniert (überschrieben) werden.
- Unter Verwendung des *Polymorphismus*-Konzeptes ist es möglich, Klassen so zu definieren, dass eine Variable sich wie Instanzen unterschiedlicher Klassen verhalten kann, ohne dass eine Konvertierung vorgenommen werden muss.

Codebeispiel Vererbung

- bekannt:

Beispiel:

```
class cFahrzeug {  
    private:  
        double laenge, hoehe, breite; int herstelljahr;  
    public:  
        void bewegen();  
};
```

Codebeispiel Vererbung

```
class cLandFahrzeug : public cFahrzeug {           //erben
private:
    int radZahl;
public:
    void fahren();
    void schieben();
};
```

```
class cWasserFahrzeug : public cFahrzeug { //erben
private:
    double tiefgang;
public:
    void anlegen();
    void ablegen();
};
```

Codebeispiel Vererbung (Fortsetzung)

```
class cAuto : public cLandFahrzeug {    // erben
private:
    double spritverbrauch;
public:
    void fahren();    // überschreibt LandFahrzeug::fahren()!
    void tanken();
    void starten();
};

class cAmphibienFahrzeug : public cAuto,
                           public cWasserFahrzeug {
    // Mehrfachvererbung
private:
    char* wassersensor;
public:
    void schottenDichtMachen();
};
```

Vererbung

```
class AT {                                // Basisklasse
    int i, j, k;
    public:
        void Doit ();
};

class BT : public AT {                    // abgeleitete Klasse
    double x, y;
    public:
        void Calculate (double arg1, int arg2);
        void Doit ();                      // Ueberschreibung
        long Doit (char *str);             // Ueberladung
};

int main () {
    BT b;

    b.Doit ("abc");
    b.Doit ();                             // Ueberschreibung wird aufgerufen
}
```

Vererbung

- Ein Mitglied in BT kann den gleichen Namen wie ein geerbtes Mitglied aus der Basisklasse AT haben. Dadurch wird das geerbte Mitglied verdeckt, und das neue Mitglied nimmt seinen Platz ein.
- Das verdeckte Mitglied ist "überschrieben".
- Auf verdeckte Mitglieder kann durch die Verwendung des sogenannten *Scope-Operators* `::` zugegriffen werden.

```
b.AT::Doit();           // Doit aus AT
```

- Eine Klasse kann *indirekt* mehrfach als Basisklasse auftreten:

```
class AT { // Basisklasse
    int i, j, k;
public:
    void Doit (void);
};
```

```
class BT : public AT { // einfache Vererbung
    double x, y;
public:
    void Calculate (double arg1, int arg2);
    long Doit (const char *str);
};
```

```
class CT : public AT { // einfache Vererbung
    char *z;
};
```

```
class DT : public BT, public CT { // Mehrfachvererbung
    int u, v;
};
```


Mehrfachvererbung

- Welche Datenmitglieder sind in einem Objekt der Klasse DT enthalten ?

Die doppelte Anwendung des Scope-Operators ::

→ ist nicht zulässig !



- Auf die Datenmitglieder von AT kann nur dort zugegriffen werden, wo eine eindeutige Auflösung ohne mehrfachen Scope-Operator möglich ist.

```
void BT::Calculate (double arg1, int arg2) {  
    /* .... Implementierung Calculate ..... */  
    AT::i += arg2;    // ????????  
}
```

Mehrfachvererbung + virtuelle Ableitungen

- Ist die doppelte Aufnahme der Datenmitglieder von AT in der Ableitung DT unerwünscht, bildet man sogenannte *virtuelle Ableitungen*.
- Dann „gewinnt“ die Version der abgeleiteten Klasse (Überschreibung wie bei linearer Ableitung).

Mehrfachvererbung + virtuelle Ableitungen

```
class AT {          // Basisklasse
    int i, j, k;
public:
    void Doit (void);
};
```

```
class BT : virtual public AT { // einfache Vererbung
    double x, y;
public:
    void Calculate (double arg1, int arg2);
    long Doit (const char *str);
};
```

```
class CT : virtual public AT { // einfache Vererbung
    char *z;
};
```

```
class DT : public BT, public CT { // Mehrfachvererbung
    int u, v;
};
```


4. Vorlesung

Konstruktoren in Klassenhierarchien

- Konstruktoren werden nicht automatisch von der Basisklasse an die Ableitungen vererbt:

```
class AT {  
    int i;  
  
public:  
    AT(int i_in) { i = i_in; }  
};
```

```
class BT : public AT {  
    int k;
```

```
public:  
    BT(int i_in, int k_in) {  
         i = i_in; // geht nicht, i ist privat  
        k = k_in;  
    }  
};
```

Konstrukturen in Klassenhierarchien

- C++ fordert in einem Konstruktor einer Ableitung den expliziten Aufruf eines Konstruktors der Basisklasse:

// Besondere Syntax für Basiskonstrukturen

```
class BT : public AT {  
    int k;  
  
public:  
    BT (int i_in, int k_in) : AT (i_in) {  
        k = k_in;  
    }  
};
```

Konstruktoren in Klassenhierarchien

- Sind mehrere Basisklassen vorhanden, werden die einzelnen Basisklassen-Konstruktoraufrufe durch Kommas getrennt:

```
class BT : public AT, public XT {
    int k;

public:
    BT (int i_in, int k_in, double f_in )
        : AT(i_in), XT(f_in) {
        k = k_in;
    }
};

int main() {
    BT b(1, 2, 1.1415);
}
```

Aufrufreihenfolge von Konstruktoren

- Grundsätzlich gilt, dass bei Aufruf eines Konstruktors für ein Objekt einer Klasse zuerst die Konstruktoren aller Basisklassen aufgerufen werden, bevor der Anweisungsblock des Konstruktors selber betreten wird.
- Sind mehrere Basisklassen vorhanden, werden diese in der Reihenfolge ihrer Deklaration bei der Ableitung initialisiert.
- Sind virtuelle Basisklassen definiert, werden diese vor den nicht-virtuellen Basisklassen initialisiert.

Destruktoren in Klassenhierarchien

- Der Destruktor wird **wie jede andere Mitgliedsfunktion** von der Basisklasse auf die Ableitung(en) vererbt.
- Definiert eine Ableitung keinen eigenen Destruktor, wird deshalb der Destruktor der Basisklasse aufgerufen, wenn das Objekt gelöscht wird.

Konstruktoren (Reloaded)

- *Konstruktoren* und *Destruktoren* sind Mitgliedsfunktionen mit speziellen Aufgaben.
- *Konstruktoren* bzw. der *Destruktor* einer Klasse werden automatisch aufgerufen, wenn Objekte erzeugt bzw. gelöscht werden.
- *Konstruktoren* können überladen werden um in unterschiedlichen Situationen unterschiedliche Initialisierungen vornehmen zu können.
- *Konstruktoren* können Vorgabeparameter (Parameter mit Vorgabewerten) enthalten.

Konstruktorenaufrufe

- Eine Klasse kann Objekte anderer Klassen als Datenmitglieder enthalten (Aggregation).
- Der *Konstruktor* der äußeren Klasse ruft dann automatisch die *Standard-Konstrukturen* für die inneren Klassen auf.
- Wir können durch die „innere Konstruktorenkaskade“ für die Übergabe der Konstruktionsparameter an die Konstruktoren der Aggregationsbestandteile sorgen.

Konstruktor und Objektdefinitionen

Prog2_Konstruktor2_ComplexT.cpp

```
//          Definition von 2 Konstruktoren
//          (kein Standard-Konstruktor )


class complexT
{
    double re, im;

public:
    complexT (double re_in)
        {re = re_in; im = 0.0;}

    complexT (double re_in, double im_in)
        {re = re_in; im = im_in;}
};

complexT c1 = 10.0;           // Konstruktor 1
complexT c2(11.0);           // Konstruktor 1
complexT c3(12.0, 13.0);     // Konstruktor 2

complexT c4 = (17.0, 18.0);  // Achtung !! Konstr. 1 !
                             // Gesamtausdruck erhält
                             // den Wert des letzten
                             // Ausdrucks
```



Codebeispiel Konstruktorenaufrufe

Prog2_Konstruktor_Aggregation1.cpp


```
class complexT {
    double re, im;
public:
    complexT (void) {re = im = 0.0;}
    complexT (double re_in) {re = re_in; im = 0.0;}
    complexT (double re_in, double im_in)
        {re = re_in; im = im_in;}
};

class TestT {
    complexT c1, c2;           // Ohne Initialisierung
    double x;
public:
    TestT (double x_in)  {x = x_in;}  // Initialis. double x
};

int main () {
    TestT t1 (10.0);
}
```

Objekte als Datenmitglieder anderer Klassen

// **Falsche Initialisierung:** Objekte als Datenmitglieder
// können nicht wie bei einer Definition initialisiert
// werden.



```
class TestT
{
    complexT c1 (x_in), c2;    // Initialisierung von c1
                             // nicht erlaubt ! Compiler
                             // erkennt c1 als Funktion
    double x;

    /* ..... weitere Mitglieder */
};
```

„innere“ Konstruktorenkaskade

```
// Richtige Initialisierung der Objekte c1 und c2 im
// Konstruktor von TestT: Aufruf Konstruktor complexT

class TestT
{
    complexT c1, c2;
    double x;

public:
    TestT (double x_in);
};

TestT::TestT (double x_in) : c1 (x_in), c2 (x_in, x_in+1)
{
    x = x_in;
}
```

Die innere Konstruktorenkaskade wird in der Literatur manchmal als „Initialisierungsliste“ bezeichnet

==> Begriffskonflikt

Initialisierung im Konstruktor

Prog2_Konstruktor_Aggregation2.cpp

Die Notation mit dem Doppelpunkt (im Konstruktor) kann auch für Basisdatentypen-Variablen verwendet werden:

```
//      class TestT
```

```
class TestT
{
    complexT c1, c2;
    double x;
public:
    TestT (double x_in);
};
```

```
TestT::TestT (double x_in) : c1(x_in), c2(x_in, x_in+1), x(x_in) {}
```


Initialisierungsliste

- Der Begriff ist in der Literatur mehrdeutig verwendet. Hier ist die Initialisierung mehrerer Objekte (*Array*) bei der Instanziierung gemeint.
- Die Initialisierungsliste muss bei der Definition der *Objekte* und nicht bei der Deklaration des *Konstruktors* angegeben werden.
- Wird ein *Array* von Objekten erzeugt, werden die einzelnen Elemente des Arrays genauso initialisiert wie bei der Erzeugung einfacher Objekte.
- Ist eine Initialisierungsliste vorhanden, werden die Konstruktoren der Array-Elemente mit den Werten der Liste aufgerufen. Sind mehr Elemente als Initialisierer vorhanden, wird für die restlichen Elemente der Standard-Konstruktor aufgerufen.

Initialisierung von Arrays

- Die explizite Angabe des Klassentyps / Konstruktors ist erforderlich:

```
complexT c11[3] = { complexT (1.0, 2.0),  
                    complexT (3.0, 4.0),  
                    complexT (5.0, 6.0) };
```

Codebeispiel Initialisierungsliste

```
// class complexT  
  
class complexT  
{  
    double re, im;  
public:  
    complexT (void) {re = im = 0.0;} // Standardkonstr.  
  
    complexT (double re_in) {re = re_in; im = 0.0;}  
  
    complexT (double re_in, double im_in)  
        {re = re_in; im = im_in;}  
};
```

Durch die Anweisung

```
complexT c7[5] = {1.0, 2.0, 3.0, 4.0};
```

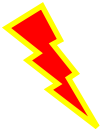
wird das Array-Element c7[4] mit (0.0, 0.0) initialisiert.

Initialisierung von Arrays

- Bei dieser Art der Initialisierung von Array-Elementen muß **immer das Gleichheitszeichen** verwendet werden und es kann nur ein Konstruktor mit **einem Argument** aufgerufen werden.
- Konstruktionen wie

```
complexT c11[3] ((1.0, 2.0), (3.0, 4.0), (5.0, 6.0));
```

sind *nicht möglich* bzw. haben nicht das gewünschte Ergebnis.



5. Vorlesung

Sichtbarkeit beeinflussen

- Mit der *Zugriffssteuerung* wird festgelegt, wie klassenfremde Funktionen auf die Mitglieder (Datenelemente und Funktionen) der Klasse zugreifen können.
- Die mit *private* gekennzeichneten Mitglieder sind nur innerhalb der Klasse sowie für Freund-Funktionen bzw. Freund-Klassen sichtbar. Steht *private* als erstes, nach dem Schlüsselwort *class*, kann *private* auch weggelassen werden (Standardeinstellung). Private Mitglieder einer Basisklasse sind in abgeleiteten Klassen nicht sichtbar.
- Die mit *protected* gekennzeichneten Mitglieder verhalten sich wie *private* Mitglieder, außer, daß sie auch in abgeleiteten Klassen sichtbar sind. Definiert man ein Mitglied einer Klasse als *protected*, ist es zwar in der Ableitung, nicht jedoch außerhalb des Ableitungsbaumes sichtbar.
- Auf die mit *public* gekennzeichneten Mitglieder kann überall im Programm zugegriffen werden.

Vererbung + Zugriffsrechte

- Mit den Mitgliedern einer Basisklasse werden auch deren Zugriffsberechtigungen vererbt. Diese Vererbung wird jedoch durch die **Zugriffsrechte der Ableitung** beeinflusst.
- Wird eine **Ableitung *private*** gemacht, werden die *protected* und *public* Mitglieder in der abgeleiteten Klasse ***private*** !
- Entsprechendes gilt für eine *protected* durchgeführte Ableitung.
- Verzichtet man auf die explizite Angabe der Zugriffsberechtigung bei der Ableitung, wird *private* angenommen.
- Durch die Ableitung können die Zugriffsberechtigungen nur **verschärft**, nicht abgeschwächt werden.

Vererbung + Zugriffsrechte

```
class AT {  
    int j, k;    // "verschlossen"  
  
    protected:    // oeffnet fuer abgeleitete Klassen  
        int i;  
  
    public:  
        void Doit (void);  
};  
  
class BT : public AT {    // Rechte-erhaltend  
    double x, y;  
  
    public:  
        void Calculate (double arg1, int arg2);  
};
```


Vererbung + Zugriffsrechte

```
void BT::Calculate (double arg1, int arg2) {  
    // ...  
    i = 0;    // ist zugaenglich da protected  
    // ...  
}
```

```
int main () {  
    BT b;  
  
    // ...  
    b.i = 1; // ist NICHT zugaenglich, nicht public  
}
```



Vererbung + Zugriffsrechte

- In einer privaten Ableitung erhalten alle geerbten Klassenmitglieder den Status *private*.

```
class AT {  
    int j, k;  
  
    protected:  
        int i;  
  
    public:  
        void Doit (void);  
};
```

```
class BT : private AT { // Rechte-einschraenkend  
    double x, y;  
  
    public:  
        void Calculate (double arg1, int arg2);  
};
```

Vererbung + Zugriffsrechte

```
int main () {  
    BT b;  
  
    b.i = 1;  
    b.j = 0;  
    b.k = 0;  
    b.Doit();  
  
    b.Calculate(3.1415, 1);  
}
```

Sind Sie mit den Anweisungen einverstanden  ?

Vererbung + Zugriffsrechte

- Redeklaration von Zugriffsberechtigungen

Oft müssen einzelne Methoden der Basisklasse für den Nutzer der Ableitung verfügbar gemacht werden. Eine Möglichkeit zur Realisierung dieser Forderung besteht in der Verwendung einer speziellen (*inline*)-Überschreibung der Methode.

Vererbung + Zugriffsrechte

// Redeklaration von Zugriffsberechtigungen auf Funktionen

```
class AT {  
    public:  
        int GetError (void) { return 0; }  
        void Doit (void) {} // wird durch private Ableitung unsichtbar  
};  
  
class BT : private AT {  
    public:  
        int GetError (void) {  
            return AT::GetError();  
        }  
};  
  
int main () {  
    BT b;  
    int i = b.GetError(); // ist zugaenglich  
}
```

Vererbung + Zugriffsrechte

- Mit der gleichen Technik können auch als *protected* deklarierte Mitglieder der Basisklasse als *public* "**redefiniert**" werden.
- C++ besitzt zudem ein Sprachmittel zur direkten Umdefinition der Zugriffsberechtigung:

Vererbung + Zugriffsrechte

```
class AT {  
  
    public:          // geht genauso mit protected: ?  
        int a;      // Kapselverletzung !  
        int GetError (void) { return 0; }  
        void Doit (void) {}  
};  
  
class BT : private AT {    // private Ableitung  
                           // macht „neue“ Kapsel  
  
    private:  
  
        /* Mitglieder von BT */  
  
    public:  
        AT::GetError;  
        AT::a;           // öffnet die Kapsel erneut  
};
```

Vererbung + Zugriffsrechte

```
int main () {  
    BT b;  
    int e = b.GetError();  
  
    b.a = 2;  
}
```


Vererbung + Zugriffsrechte

- Zur Umdefinition reicht es aus, die Bezeichner in der Ableitung unter dem entsprechenden Schlüsselwort (hier *public*) erneut aufzuführen.

Syntax bzw. Semantik Aspekte:

- Funktionen ohne Argumentenliste, Klammern und Typ
 - ➔ **1** Umdefinition reicht für **alle** Überladungen -- flexibler !
 - Die Redeklaration kann nur auf die ursprüngliche Deklaration zurückgestellt werden.
- Verwendung: privat abgeleitete Klasse stellt nur einen Teil der Schnittstelle zur Verfügung

Freund-Funktionen

- Eine *Freund-Funktion* ist eine Funktion, die zwar kein Mitglied der eigenen Klasse ist, aber trotzdem auf alle (auch private) 'Mitglieder' der Klasse zugreifen kann.
- Eine *Freund-Funktion* muß innerhalb der Definition der Klasse mit dem Schlüsselwort *friend* bekannt gemacht werden.
- Freund-Funktionen können freie Funktionen oder Methoden anderer Klasse sein.

Codebeispiel *friend*-Konstrukt

- Globale Funktion GetRe() als Freund-Funktion

```
// class complexT
```

```
class complexT
```

```
{
```

```
private:
```

```
    double re, im;
```

```
    friend double GetRe (complexT s1);
```

```
public:
```

```
    void set (double re_in, double im_in) {re=re_in; im=im_in;}
```

```
    void print (void);
```

```
};
```

```
double GetRe (complexT s1)
```

```
{
```

```
    return (s1.re);
```

```
}
```

Codebeispiel *friend*-Konstrukt

- Ist eine Funktion eine Klassenfunktion (Methode), kann sie ebenfalls als Freund deklariert werden. Die zugehörige Klasse muß jedoch bereits definiert oder deklariert sein.

```
class complexT;           // complexT muß vor specialT
                           // deklariert werden

// class specialT

class specialT
{
private:
    double f1, f2;
    void interFace (complexT c);    // hier nur Prototyp

    // ..... weitere Mitglieder
};
```

Codebeispiel *friend*-Konstrukt (Fortsetzung)

```
//          class complexT

class complexT
{
private:
    double re, im;

    friend void specialT::interFace (complexT c);

    // ..... weitere Mitglieder
};

// SpecialT::interFace darf erst nach complexT
// implementiert werden

void specialT::interFace (complexT c)
{
    f1 = c.re;
    f2 = c.im;
}
```

friend-Konstrukt (Fortsetzung)

- Die Funktion *interFace* ist *private*, d.h. sie kann nur von anderen Mitgliedsfunktionen von *specialT* aufgerufen werden. Nutzer der Klassen *complexT* oder *specialT* können auf *interFace* nicht zugreifen.
- *interFace* kann sowohl auf die Mitglieder der eigenen Klasse als auch auf die von *complexT* zugreifen.

-
- Außer Funktionen können auch ganze Klassen als Freund-Klassen definiert werden.
 - Freund-Deklarationen werden nicht vererbt.

Inline-Methoden

- *Inline-Methode*: In C++ kann eine Methode als *inline* deklariert werden.
- Inline-Methoden werden nicht aufgerufen wie andere Methoden oder Funktionen, sondern durch den Compiler an der „Aufrufstelle“ in den Code einkopiert (wenn möglich).
- Die Methoden einer Klasse werden innerhalb der Klassendefinition i. d. R. nur deklariert. Die Definition der Methode kann jedoch gleich mit angegeben werden; dadurch wird die Methode automatisch *inline*.
- Die *inline*-Spezifikation kann auch extern erfolgen.
- Zu beachten wenn die Klassendefinition und die Implementierung der Methoden auf verschiedene Quellendateien verteilt sind.

Codebeispiel Methodendeklaration extern *inline*

```
// class complexT

class complexT
{
private:
    double re, im;

public:
    void set (double re_in, double im_in)           // inline
        { re = re_in; im = im_in;}

    void print (void);                             // Deklaration von print
};

// Externe Definition von print

inline void complexT::print (void)
{
    cout << "( re : " << re << ", im : " << im << " )" << endl;
}
```


6. Vorlesung

Klassen in zusammengesetzten Datentypen

Klassen können unbeschränkt zur Definition von zusammengesetzten Datentypen wie *arrays*, *structs* oder anderen Klassen verwendet werden.

Beispielklasse:


```
//          class complexT

class complexT
{
private:
    double re, im;

    //          .....      weitere Mitglieder
};
```

Grenzen der Typkonstruktion

Eine Klasse kann kein Mitglied vom eigenen Typ definieren:

```
class wrongT    // Compilierung nicht erfolgreich
{
private:
    double f;
    char * name;
     wrongT W;    // Objekt der eigenen Klasse
    // ..... weitere Mitglieder
};
```

Zeiger auf den eigenen Typ

Eine Klasse kann aber einen Zeiger auf den eigenen Typ als Mitglied definieren:

```
class rightT
{
private:
    double f;
    char * name;
    rightT * r;    // Zeiger auf Objekt der eigenen Klasse

    // ..... weitere Mitglieder
};
```

Mittels Zeiger auf Objekte der eigenen Klasse können lineare Listen oder rekursive Datenstrukturen aufgebaut werden.

Übungsfrage

Warum kann eine Klasse nicht Mitglied ihrer eigenen Datendefinition sein („rekursive Definition“), wohl aber darf ein Zeiger auf ein Objekt ihres Typs in ihrer Datendefinition vorkommen ?

- A) Weil die Syntax von C++ es verbietet
- B) Ein Zeiger hat eine definierte Länge, ein Objekt aber nicht, weil zu diesem Zeitpunkt es noch nicht fertig definiert ist.
- C) Eine Klasse kennt sich selbst nicht
- D) Die doppelte Nennung des Klassennamens ist verboten

Objekte als Parameter von Funktionen

- Objekte können wie normale Datenelemente als Parameter an Funktionen übergeben und von diesen auch zurückgeliefert werden (C++ lässt im Gegensatz zu C zusammengesetzte Typen als Ergebnistypen von Funktionen zu).
- Beispiel für Parameterübergabe:

```
bool Iszero (complexT c, double Epsilon = 0.0001) {  
    return ((abs(c.re) <= Epsilon) && (abs(c.im) <= Epsilon));  
}
```

Was ist mit den Zugriffsrechten auf die Attribute von c?

Objekte als Parameter von Funktionen

Der Compiler codiert folgende Schritte:

1. Erzeugen eines Objektes `c` vom Typ `complexT` auf dem Stack
2. Aufruf des Konstruktors für `c`
3. Kopieren der übergebenen Datenelemente in den Datenbereich von `c`
4. Aufruf des Destruktors für `c` (nach Ende der Funktion)

Referenztyp

- In C gibt es nur die Möglichkeit zur Wertübergabe (call by value).
- C++ bietet eine Lösung in Form des Referenztyps:

```
// j ist während der Laufzeit von main eine
// Referenz auf i

int main () {

    int i;
    int& j = i;    // Referenz wird mit Hilfe des
                  // Referenz-Operators & definiert

    i = 5;
    j++;

    cout << "i hat den Wert " << i << endl;
}
```

- *i* und *j* sind unterschiedliche Namen für das gleiche Datenobjekt; *j* bleibt während seiner gesamten Lebenszeit fest mit *i* verbunden.

Referenztyp

- Die Referenz ist ein "Alias-Name"
- ohne die Dereferenzierungssyntax
- und sicher ! (ein Zeiger kann verloren gehen).
- Die immer erforderliche Initialisierung einer Referenz kann beim Compilieren oder beim Linken erfolgen:

```
int i;  int &j = i;    // Init. beim Compilieren
```

- Ein Referenztyp wird zur Parameterübergabe eingesetzt (call by reference):

```
void auswechsel (int &a, int &b) {  
    int    help = a;  
  
    a = b;  
    b = help;  
}
```

Referenztyp

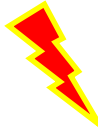
- Noch ein Beispiel:

```
void inc (int &i) { /* Alternativschreibweise: int& i */  
    i++;  
}  
  
int main () {  
    int a = 1;  
    int b = 10;  
  
    inc (a);  
    inc (b);  
    inc (b);  
  
    cout << "Werte von a und b: " << a << b << endl;  
}
```

- Ist der Funktionsaufruf abgearbeitet, hört *i* (und damit die Bindung an *a*) auf zu existieren. Bei weiteren Funktionsaufrufen kann (ein neues) *i* dann an andere Variablen gebunden werden.

Referenztyp

Einschränkungen bei Referenztypen:

- Können nur bei der Definition gesetzt, nicht wieder umgesetzt werden.
- Es sind keine Zeiger auf Referenzen möglich.
- Es gibt keine Referenzen vom Typ `void&` .

- Es gibt keine Referenzen auf Referenzen (`int & &`).
- Es gibt keine Arrays aus Referenzen.

Referenztyp

- Bei der Übergabe eines Objekts als Argument (call by value), wird die *lokale Kopie* mittels des Kopier-Konstruktors initialisiert sowie am Ende der Funktion durch den Destruktor wieder zerstört.
- Eine solche *explizite Kopie* sollte nur dann erstellt werden, wenn die Funktion die Kopie verändert, das Original aber unverändert bleiben soll (muss).
- Die *Referenz* bewirkt, dass das *Original* verwendet wird, es entsteht kein Umspeicherverlust.
- Wenn man den Umspeicherverlust vermeiden möchte, das Objekt aber nicht verändern möchte, kann die Referenz „*const*“ angegeben werden.

Codebeispiel 1 Referenztyp

```
//          class complexT

class complexT {
    double re, im;
public:
    complexT (void) {re = im = 0.0;}
    complexT (double re_in) {re = re_in; im = 0.0;}
    complexT (double re_in, double im_in)
        {re = re_in; im = im_in;}

    friend bool IsZero(complexT &c, double Epsilon=0.0001);
};
```

(bekannte Klassendefinition)

Codebeispiel 1 Referenztyp

```
// IsZero

bool IsZero (complexT &c, double Epsilon = 0.0001) {
    return ((abs(c.re) <= Epsilon) && (abs(c.im) <= Epsilon));
}


// main

int main () {

    complexT c1 (0, 0), c2 (10, 11);
    bool i = IsZero (c1);
    bool j = IsZero (c2);
}
```

Codebeispiel Referenztyp als *const*

Wird ein Parameter an eine Funktion als *const* übergeben, kann die Funktion den Wert des Parameters nicht verändern (*read-only*).

```
bool IsZero (const complexT &c, double Epsilon = 0.0001) {  
    return ((abs(c.re) <= Epsilon) && (abs(c.im) <= Epsilon));  
}
```

```
int main () {  
    complexT c1 (0, 0), c2 (10, 11);  
    bool i = IsZero (c1);  
    bool j = IsZero (c2);  
}
```

Referenzen als Klassenmitglieder


- In C++ kann eine Referenz ein Datenmitglied einer Klasse sein; Initialisierung erfolgt im Konstruktor mit einer speziellen Notation.

```
class DispT {  
  
    int x,y;        // Bildschirmkoordinaten  
    char *&msg;     // Referenz auf eine Zeichenkette  
                   // (kein Zeiger)  
  
public:  
    DispT (int x_in, int y_in, char *&msg_in);  
    void print (void);  
};
```


Referenzen als Klassenmitglieder

- Initialisierung der Referenz im Konstruktor !
(analog innere Kaskade)

```
DispT::DispT (int x_in, int y_in,  
              char *&msg_in) : msg (msg_in) {  
  
    x = x_in;  
    y = y_in;  
    /* msg = msg_in; wäre eine nicht korrekte  
       Initialisierung !  
    */  
}
```



- Enthält eine Klasse mehrere Referenzen, sind die einzelnen Initialisierer durch Kommas zu trennen.

Referenzen als Funktionsrückgaben

- Referenzen, als Ergebnis eines Funktionsaufrufes, können nur an globale bzw. statische Variablen zurückgeliefert werden.

```
int &Doit (void) {  
    int i = 7;  
    return i; // Falsch! i ist dann weg!  
}
```



```
int &Doit (void) {  
    static int i = 7;  
    return i; // so gehts  
}
```

Referenzen als Funktionsrückgaben

```
char *&Doit (int i) {  
    static char *msg1 = "Dies ist ein String";  
    static char *msg2 = "Dies ist auch ein String";  
  
    if (i == 1)  
        return msg1;  
    else  
        return msg2;  
}  
  
int main () {  
    cout << Doit (1) << endl;  
}
```

- Funktionen, die Referenzen zurückgeben, können als Ziel einer Wertzuweisung stehen.

```
Doit (1) = "Eine andere Zeichenkette";
```

Codebeispiel Referenzen

- *Beispielprogramm*

Prog2_Bruch_noReferenz.cpp

Prog2_Bruch_Referenz.cpp

live Demo

Kopierkonstruktor

(vormals Teil08)

Der Kopierkonstruktor:

```
ClassT::ClassT( const ClassT & );
```

Der Kopierkonstruktor wird immer dann verwendet, wenn ein Objekt mit einem anderen Objekt initialisiert wird.

Kopierkonstruktor Beispiel

Explizite Initialisierung:

```
#include <string.h>

// class MessageT

class MessageT
{
    char* p; // Der Zeiger auf den Text der Meldung

public:
    MessageT (void);
    MessageT (char* p_in);
    MessageT (const MessageT &msg_in);
    ~MessageT (void);
};
```

Kopierkonstruktor Beispiel

```
MessageT::MessageT (void) { p = Null; }
```

```
MessageT::MessageT (char* p_in) {  
    if (!(p = new char [strlen (p_in)+1])) {  
        // p == Null ==> Fehler, kein Speicher  
        return;  
    }  
    strcpy (p, p_in);  
}
```

```
MessageT::~~MessageT (void) {  
    if (p)  
        delete [] p;  
  
    p = Null;  
}
```

Kopierkonstruktor Beispiel

```
// MessageT Kopierkonstruktor
```

```
MessageT::MessageT (const MessageT &msg_in) {  
  
    if ( !(p = new char[strlen(msg_in.p) +1]) ) {  
        // kein Speicher  
        return;  
    }  
  
    strcpy(p, msg_in.p);  
}
```

Der Kopierkonstruktor kann nur mit einem Referenz-Parameter funktionieren, weil er mit call by value eine Kopie erstellen müsste, dazu wäre der Kopierkonstruktor aufzurufen, der eine Kopie

Kopierkonstruktor Beispiel

- Der Kopier-Konstruktor wird nur bei der Initialisierung von Objekten, nicht jedoch bei Zuweisungen vom Compiler eingebunden.

```
int main () {  
  
    MessageT msg1("Betriebssystemfehler");  
    MessageT msg2=msg1; // Aufruf Kopierkonstruktor  
    MessageT msg3;  
  
    msg3 = msg1; // Kein Aufruf von Kopierkonstr.  
                // sondern: Zuweisungsoperator  
                :  
                :  
}
```

7.Vorlesung

Überladen von Funktionen

- In C++ kann der gleiche Funktionsname für unterschiedliche Funktionen verwendet werden.
- Daher kann es innerhalb eines Gültigkeitsbereiches mehrere Funktionen gleichen Namens geben, die sich aber in Anzahl und/oder Typ der Parameter unterscheiden müssen.
- Welche Funktion bei einem Funktionsaufruf zu verwenden ist, erkennt der Compiler an Typ und Anzahl der Parameter (Signatur der Funktion).
- Der Compiler sucht nach der Funktion mit passendem Namen und passender Argumentenliste (Signatur).
- Beispiel: Funktion *add* für Integer-Zahlen und für komplexe Zahlen:

Überladen von Funktionen

(Prog2_ueberlad_add2.cpp)

```
// Die Funktion add() wird fuer complexT redefiniert

// class complexT
class complexT {

    friend complexT add (complexT lhs, complexT rhs);

    double re, im;

public:
    complexT (void) {re = im = 0.0;}
    complexT (double re_in) {re = re_in; im = 0.0;}
    complexT (double re_in, double im_in)
        {re = re_in; im = im_in;}

    void write (void) {
        cout << "Realteil: " << re <<
            " Imaginaerteil: " << im << endl;
    }

};
```

Überladen von Funktionen

```
// add für int-Argumente
```

```
int add ( int lhs, int rhs) {  
    return (lhs + rhs);  
}
```

```
// add für complexT-Argumente
```

```
complexT add (complexT lhs, complexT rhs) {  
    complexT erg;  
  
    erg.re = lhs.re + rhs.re;  
    erg.im = lhs.im + rhs.im;  
  
    return (erg);  
}
```

Überladen von Funktionen

```
int main () {  
    int r1 = 1;  
    int r2 = 2;  
    int r3;  
  
    complexT x1(10,11);  
    complexT x2(1,2);  
    complexT x3;  
  
    r3 = add(r1,r2);    // add für integer  
    x3 = add(x1,x2);    // add für complexT  
  
    cout << "Integer-Addition: " << r3 << endl;  
    x3.write();  
}
```

Überladen von Funktionen

(Prog2_ueberlad_add2.cpp)

```
// class complexT
class complexT {
    friend complexT add (complexT lhs, complexT rhs);
    friend complexT add (complexT lhs, double rhs);
};

// Die Funktion add() wird fuer complexT redefiniert
// add fuer complexT und double

complexT add(complexT lhs, double rhs) {
    return(complexT(lhs.re + rhs, lhs.im));
}
```

- Es können alle möglichen Kombinationen von Parametertypen mit jeweils 'einer' Funktion abgedeckt werden.

Überladen von Funktionen

```
//      main

int main () {
    complexT x1(10,11);
    complexT x4;

    x4 = add(x1,5.0);

    x4.write();
}
```


Überladen von Methoden

- Mitgliedsfunktionen können wie normale Funktionen überladen werden:
 - (*Prog2_ueberlad_mfunk1.cpp*)

// Die write-Funktion von complexT wird überladen

// class complexT

```
class complexT {
    double re, im;

public:
    complexT (void) {re = im = 0.0;}
    complexT (double re_in) {re = re_in; im = 0.0;}
    complexT (double re_in, double im_in)
        {re = re_in; im = im_in;}
    void write (void);
    void write (int x, int y);
};
```

Überladen von Methoden

```
//      write ohne Argument
```

```
void complexT::write (void) {  
    cout << "Realteil: " << re <<  
        " Imaginaerteil: " << im << endl;  
}
```

```
//      write mit Koordinaten auf dem Bildschirm
```

```
void complexT::write (int x, int y) {  
    cursorPosition (x,y);  
    write ();  
}
```

Überladen von Methoden

```
int main () {  
    complexT x1(10,11);  
  
    clrscr();  
  
    x1.write();  
    x1.write(12, 21);  
}
```

- Eine Methode kann nur überladen werden, wenn die neue Methode im gleichen Gültigkeitsbereich wie die zu überladende Methode definiert wird.

Überladen von Operatoren

(vormals Teile 10+11)

- *Überladen* von Operatoren: In C++ kann (fast) jeder Operator für benutzerdefinierte Datentypen neu implementiert werden.
- Damit lässt sich zum Beispiel ein Vergleichsoperator „>“ für eine Klasse „Schiffe“ erstellen, der z.B. anhand der Wasserverdrängung, der Länge und der Passagierzahl einen direkten Vergleich erlaubt:

```
if (Titanic > QueenElizabethII) { ... }
```

- Beispiel: Implementierung der Operatorfunktion für den Additionsoperator „+“ für komplexe Zahlen:

(Prog2_ueberlad_op+1.cpp)

Überladen von Operatoren

```
//      class complexT

class complexT {
    friend complexT operator + (complexT lhs, complexT rhs);

    double re, im;
public:

    complexT (void) { re = im = 0.0; }
    complexT (double re_in) { re = re_in; im = 0.0; }
    complexT (double re_in, double im_in)
        { re = re_in; im = im_in; }

    void write (void);
    void write (int x, int y);
};
```

Überladen von Operatoren

// Operator + für komplexe Zahlen

```
complexT operator + (complexT lhs, complexT rhs) {  
    return complexT (lhs.re + rhs.re, lhs.im + rhs.im);  
}
```

- In *main()* ist nun

```
x3 = x1 + x2;
```

- gleichbedeutend mit

```
x3 = operator + (x1, x2);
```

- „Professionelle“ Implementierung des „+“ Operators für komplexe Zahlen vermeidet die Erzeugung von temporären Objekten (Referenz!):
(Prog2_ueberlad_op+2.cpp)

```
//    class complexT

class complexT {
    friend complexT &operator + (const complexT &lhs,
                                const complexT &rhs);

    double re, im;
public:
    complexT (void) { re = im = 0.0; }
    complexT (double re_in) {re = re_in; im = 0.0; }
    complexT (double re_in, double im_in)
        { re = re_in; im = im_in; }

    void write (void);
    void write (int x, int y);
};
```

Überladen von Operatoren

```
//      Operator + für komplexe Zahlen
//
complexT &operator + ( const complexT &lhs,
                      const complexT &rhs) {

    static complexT CalcBuffer;

    CalcBuffer.re = lhs.re + rhs.re;
    CalcBuffer.im = lhs.im + rhs.im;

    return CalcBuffer;
}
```



```
//-----  
//    main  
//  
int main () {  
  
    complexT x1(10,11);  
    complexT x2(1, 2);  
    complexT x3;  
    complexT x4;  
  
    x3 = x1 + x2;  
    x3.write();  
  
    // Der Puffer arbeitet auch bei Kettenrechnung korrekt  
  
    x3 = x1 + 1 + x2 + 3;  
    x3.write();  
  
    x4 = x1 + x2 + x3;  
    x4.write();  
}
```

Operator als Methode der Klasse

- Wenn eine Operatorfunktion als Mitgliedsfunktion einer Klasse definiert wird, hat sie automatisch ein „Argument“ vom Typ der Klasse, nämlich sich selbst.
- Beispiel: Definition der Operatorfunktion für den Additionsoperator „+“ als Mitgliedsfunktion von complexT:

(Prog2_ueberlad_op+mf1.cpp)

Operator als Methode der Klasse

```
//      class complexT

class complexT {
    double re, im;

public:
    complexT operator + (const complexT rhs);

    /* ... weitere Mitglieder ... */
};

//      Operator + für komplexe Zahlen

complexT complexT::operator + (const complexT rhs) {
    return complexT (re + rhs.re , im + rhs.im);
}
```

Operator als Methode der Klasse

```
int main () {  
  
    double r1 = 1.0;  
    double r2 = 2.0;  
    double r3;  
  
    complexT x1(10,11);  
    complexT x2( 1, 2);  
    complexT x3;  
  
    r3 = r1 + r2;  
    x3 = x1 + x2;  
    x3.write();  
}
```

Operator als Methode der Klasse

- Obwohl operator + in complexT mit nur einem Argument definiert wurde, wird er im Hauptprogramm mit zwei Argumenten aufgerufen.

x3 = x1 + x2;

ist identisch mit der Anweisung

x3 = x1.operator + (x2);

Aufruf der Mitgliedsfunktion „operator + ()“

- Die aktuelle Instanz wird grundsätzlich als zusätzlicher (erster) Parameter der Operatorfunktion interpretiert.

- „Professionelle“ Definition der Operatorfunktion für den Additionsoperator "+" als Methode von complexT:
(Prog2_ueberlad_op+mf2.cpp)

```
//    class complexT
class complexT {

    /*    ...    weitere Mitglieder    ...    */

    complexT &operator  + (const complexT &rhs);

};

//    Operator  +    für komplexe Zahlen
complexT &complexT::operator + (const complexT &rhs) {
    static complexT CalcBuffer;

    CalcBuffer.re = re + rhs.re;
    CalcBuffer.im = im + rhs.im;

    return CalcBuffer;
}
```

Methoden als „const“ kennzeichnen

- Neben der Möglichkeit, Parameter einer Methode (oder Funktion) als „const“ zu kennzeichnen, können auch ganze Methode als „const“ gekennzeichnet werden.
- Damit kann der Compiler in Methoden (oder Funktionen), die const-Methoden (oder Funktionen) aufrufen, sicher sein, dass mit dem Aufruf keine beteiligten Objekte verändert werden und einen const-Parameter verlässlich schützen:

complexT &operator + (const complexT &rhs) const;

=> „const“ schützt hier auch das this-Objekt (lhs) vor Veränderung

Überladen von Operatoren

- Beschränkungen

Mit Ausnahme der folgenden Operatoren können in C++ alle C - Operatoren und C++ - Operatoren überladen werden:

`.` `.*` `::` `?:` `#` `##`

Die folgenden Operatoren können nur als Methoden von Klassen redefiniert werden:

`=` `[]` `()` `->`

Überladen von Operatoren

Weiterhin gilt:

- Die Priorität der Operatoren kann nicht geändert werden
- Die Stelligkeit kann nicht geändert werden
- Operatoren die sowohl einstellig als auch zweistellig sind, gelten für die Redefinition als unterschiedliche Operatoren
- Wird der Inkrement- oder Dekrementoperator überladen, wird zwischen Präfix- und Postfixnotation unterschieden über eine trickreiche Konstruktion mit einem int Pseudo-Parameter

Beispiel: Überladen von Operatoren

- Die Operatoren ++ und -- werden überladen, einmal als Methode und einmal als Freundfunktion:
 - (*Prog2_ueberlad_op++--3.cpp*)

Beispiel: Überladen von Operatoren

```
// C++-Demoprogramm fuer ueberladene Operatoren
// Die Operatoren "++" und "--" werden fuer complexT ueberladen
// Postfix und Praefix moeglich,
// "--" als Freundfunktionen,
// "++" als Mitgliedsfunktionen.

#include <iostream>
using namespace std;

class complexT {
    friend complexT operator -- (complexT &c1); // Praefix-Dekr.
    friend complexT operator -- (complexT &c1, int); // Postfix-Dekr.
    double re, im;
public:
    complexT (void) {re = im = 0.0;}
    complexT (double re_in) {re = re_in; im = 0.0;}
    complexT (double re_in, double im_in) {re = re_in; im = im_in;}

    complexT operator ++ (void); // Praefix-Inkrement (++x)
    complexT operator ++ (int); // Postfix-Inkrement (x++)

    void write (const char * text) {
        cout << "Realteil: " << re << " Imaginaerteil: " <<
            im << " (" << text << ") \n\n";
    }
};
```

Beispiel: Überladen von Operatoren

```
complexT operator -- (complexT& c) {    // Praefix-Dekrement (--x) als Friend-F.  
    c.re -= 1.0;  
    c.im -= 1.0;  
    return c;  
}  
  
complexT operator -- (complexT& c, int) { // Postfix-Dekrement (x--) als Friend-F.  
    complexT tmp = c;  
    c.re -= 1.0;  
    c.im -= 1.0;  
    return tmp;  
}  
  
complexT complexT::operator ++ (void) { // Praefix-Inkrement (++x) als Methode  
    re += 1.0;  
    im += 1.0;  
    return *this;  
}  
  
complexT complexT::operator ++ (int) { // Postfix-Inkrement (x++) als Methode  
    complexT tmp = *this;  
    re += 1.0;  
    im += 1.0;  
    return tmp;  
}
```

Zugriff auf das Objekt über *this* (impliziter erster Parameter)

Beispiel: Überladen von Operatoren

```
int main (void) {  
    complexT x1(10,11);  
    complexT x2;  
    complexT x3;  
  
    x1.write ("x1 neu");  
  
    x1++;  
    x1.write ("x1 nach x1++");  
  
    x2 = x1++;  
  
    x1.write ("x1 nach x2 = x1++");  
    x2.write ("x2 nach x2 = x1++");  
  
    x3 = ++x2;  
    x2.write ("x2 nach x3 = ++x2");  
    x3.write ("x3 nach x3 = ++x2");  
  
    --x1;  
    x1.write ("x1 nach --x1");  
  
    x2 = x1--;  
  
    x1.write ("x1 nach x2 = x1--");  
    x2.write ("x2 nach x2 = x1--");  
  
    return 0;  
}
```

Beispiel: Überladen von Operatoren

Ausgabe des Programms:

```
Realteil: 10 Imaginaerteil: 11 (x1 neu)
Realteil: 11 Imaginaerteil: 12 (x1 nach x1++)
Realteil: 12 Imaginaerteil: 13 (x1 nach x2 = x1++)
Realteil: 11 Imaginaerteil: 12 (x2 nach x2 = x1++)
Realteil: 12 Imaginaerteil: 13 (x2 nach x3 = ++x2)
Realteil: 12 Imaginaerteil: 13 (x3 nach x3 = ++x2)
Realteil: 11 Imaginaerteil: 12 (x1 nach --x1)
Realteil: 10 Imaginaerteil: 11 (x1 nach x2 = x1--)
Realteil: 11 Imaginaerteil: 12 (x2 nach x2 = x1--)
```

freiwillige Übung

- Erstellen Sie zwei weitere Versionen des Programms:
 - a) alle 4 Operatorfunktionen als Freundfunktionen
 - b) alle 4 Operatorfunktionen als Mitgliedsfunktionen

Überladung der Ein- / Ausgabeoperatoren

- Für die bekannte Klasse complexT:

```
// Ueberladung << als friend ! (wegen Parameter 1)
```

```
friend ostream& operator << (ostream& o, const complexT &c) {  
    o << "complexT, re = " << c.re << ", im = " << c.im;  
    return o;  
}
```

- Verwendung:

```
int main() {  
    complexT c1 (1.0, 2.0);  
  
    cout << "Hier kommt c1: " << c1 << endl;  
    return 0;  
}
```


Überladen der Zuweisungsoperatoren

- Die Operatoren dieser Gruppe sind:

= *= /= %= += -= >>= <<= &= ^= |=

- Der Zuweisungsoperator für die Klasse MessageT wird überladen:
 - (*ueberlad_opzuweis1.cpp*)
 - (*ueberlad_opzuweis2.cpp*)

Zuweisungen mit Objekten

- Objekte können wie Datenelemente der Standardtypen zugewiesen werden:

```
complexT c1 (12,13);  
complexT c2;
```

```
c2 = c1;
```

- Der Compiler löst die Objektzuweisung in Zuweisungen für einzelne Datenelemente auf.

```
c2.re = c1.re;  
c2.im = c1.im;
```

- Enthält eine Klasse zusammengesetzte Datenstrukturen, werden auch diese Komponente für Komponente kopiert.
- Enthält eine Klasse einen Zeiger auf einen Speicherbereich, zeigen nach einer Zuweisung die Zeiger beider Objekte auf den gleichen Speicherbereich. (Speicherbereich wird **nicht** kopiert!)

Zuweisungen mit Objekten

Beispiel:

Die folgende Klasse *MessageT* speichert einen String:

```
#include <string.h>

//      class MessageT
//
class MessageT
{
    char* p;      // Der Zeiger auf den Text der Meldung

public:
    MessageT (void);
    MessageT (char* p_in);
    ~MessageT (void);
};
```

Zuweisungen mit Objekten

```
MessageT::MessageT (void) { p = Null; }
```

```
MessageT::MessageT (char* p_in) {  
    if (!(p = new char [strlen (p_in)+1])) {  
        // p == Null ==> Fehler, kein Speicher  
        return;  
    }  
    strcpy (p, p_in);  
}
```

```
MessageT::~~MessageT (void) {  
    if (p)  
        delete [] p;  
  
    p = Null;  
}
```

```
:  
MessageT msg1 ("Fehler aufgetreten");  
MessageT msg2 ("Alles ist gut");
```

Zuweisungen mit Objekten

Anfangssituation:

msg1

p: -----> "Fehler aufgetreten"

msg2

p: -----> "Alles ist gut"

Zuweisungen mit Objekten

Zuweisung:

```
msg2 = msg1;
```

msg1

p: -----> "Fehler aufgetreten"

msg2

p: -----> "Fehler aufgetreten"

Zuweisungen mit Objekten

ACHTUNG!

Sind *msg1* u. *msg2* in einer Funktion definiert, werden am Ende der Funktion die Destruktoren aufgerufen, was zu einer doppelten Freigabe des Speichers führen würde !

```
MessageT::~~MessageT (void) {  
    if (p)          // hilft das wirklich    ?  
        delete [] p;  
    p = Null;  
}
```



Frage: Was passiert mit "Alles ist gut" ?

Überladen der Zuweisungsoperatoren

```
//      class MessageT

class MessageT {
    char* p; // Zeiger auf den Text der Meldung

public:
    MessageT (void);
    MessageT (char* p_in);
    MessageT (const MessageT &msg_in); // Kopierkonstr.
    ~MessageT (void);

    void Show (void);

    MessageT operator = (MessageT msg_in);
};
```


Überladen der Zuweisungsoperatoren

```
// MessageT Kopierkonstruktor
```

```
MessageT::MessageT (const MessageT &msg_in) {  
  
    if ( !(p = new char[strlen(msg_in.p) +1]) ) {  
        // p=NULL kein Speicher  
        return;  
    }  
    else {  
        strcpy (p, msg_in.p);  
    }  
}
```

Überladen der Zuweisungsoperatoren

```
// MessageT operator =
```

```
MessageT MessageT::operator = (const MessageT &msg_in) {
```

```
    if (p) {                // erst mal aufräumen
        delete [] p;
    }
    if ( !(p = new char[strlen(msg_in.p) +1]) ) {
        // p == NULL kein Speicher
        return *this;
    }
    else {
        strcpy(p, msg_in.p);
        return *this;
    }
}
```

Überladen der Zuweisungsoperatoren

```
int main () {           // Version mit Objektdefinition

    MessageT msg1 ("USB-Stick fehlt"),
              msg2 ("USB-Stick nicht lesbar");

    msg1.Show();
    msg2.Show();

    cout << "Zuweisung jetzt" << endl;
    msg1 = msg2;

    msg1.Show();
    msg2.Show();

}
```

Überladen der Zuweisungsoperatoren

Prog2_ueberlad_opzuweis2.cpp

```
int main () {           // Version mit Zeigern und new
    MessageT *msg1p = new MessageT
                        ("USB-Stick fehlt"),
    *msg2p = new MessageT
                        ("USB-Stick nicht lesbar");

    msg1p -> Show();
    msg2p -> Show();

    cout << "Zuweisung jetzt" << endl;
    *msg1p = *msg2p;

    msg1p -> Show();
    msg2p -> Show();
    delete msg1p;
    delete msg2p;
}
```

Was fehlt ?

Übungsfrage

Kann der Zuweisungsoperator durch „const“ zu einer konstanten Methode gemacht werden ?

- A) ja, weil das rechte Hand Seite Objekt nicht verändert wird
- B) ja, weil keine weiteren Methoden oder Funktionen aufgerufen werden dürfen
- C) nein, weil das aktuelle Objekt geändert wird
- D) nein, weil sich beide Operanden ändern, indem sie sich einander in den Attributwerten annähern

Überladen des Subscript - Operators []

- Die Operatorfunktion für Subscript-Operator [] wird überladen, um Zugriffe außerhalb der Feldgrenzen zu erkennen:

*(Prog2_ueberlad_opsubscript2.cpp,
Prog2_ueberlad_opsubscript1.cpp,
Prog2_ueberlad_opcubscript1b.cpp)*

Überladen des Subscript - Operators []

```
//      cIntArr

class cIntArr {
    int *p;    // Zeiger auf das eigentliche Feld
    int lim;   // Größter möglicher Index + 1

public:
    cIntArr (int n_lim);
    ~cIntArr (void);

    int &operator [ ] (int Index);
};
```

Überladen des Subscript - Operators `[]`

```
//      cIntArr Konstruktor

cIntArr::cIntArr (int n_lim) {

    p = new int [n_lim];

    if (p) {

        lim = n_lim;

        // evtl. Feld initialisieren

    }
    else
        lim = 0;
}
```


Überladen des Subscript - Operators `[]`

```
//      cIntArr Destruktor

cIntArr::~cIntArr (void) {

    if (p) {
        delete [] p;
        p = NULL;
        lim = 0;
    }
}
```

Überladen des Subscript - Operators []

```
//      cIntArr operator [ ]

int &cIntArr::operator [ ] (int Index) {

    if ((Index >= 0) && (Index < lim))
        return p[Index];

    //      hier kommt die Fehlerbehandlung

    cerr << "\ncIntArr : Zugriff ausserhalb Bereich"
          << " (Ist: " << Index << " /Max : "
          << lim-1 << ")\n";
    exit(1);
}
```

Überladen des Subscript - Operators []

```
//      main

int main () {
    cIntArr a(10);
    int i;

    for (i=0; i < 10; i++)
        a[i] = i * i; // a.operator [] (i) = i * i;

    for (i=0; i < 10; i++)
        cout << "\nindex : " << i << " Wert : "
              << a[i] << endl;

    for (i=0; i <= 10; i++) { // Verletzung Array-Grenze
        a[i] = i * i;
        cout << "\nindex : " << i << " Wert : "
              << a[i] << endl;
    }
}
```

Überladen des Funktionsaufruf-Operators ()

- Wird der Funktionsaufruf-Operator für eine Klasse überladen, kann man Objekte dieser Klasse wie Funktionen verwenden.
- Das Objekt verhält sich wie eine Funktion, kann aber auf sämtliche Attribute zugreifen, nicht nur auf die übergebenen Parameter.
- Beispiel:
 - *(Prog2_ueberlad_opaufruf1.cpp)*

Überladen des Funktionsaufruf-Operators ()

```
//      AT

class AT {
    int i;

public:
    AT (int i_in)  { i = i_in; }

    double operator ( ) (double faktor)  { return i*faktor; }
};

int main () {
    AT a(15);

    double j = a(2.0);
    cout << "j = " << j << endl;
}
```

8. Vorlesung

Typumwandlungen

(vormals Teil07)

- Zuweisungen zwischen Objekten unterschiedlicher Klassen
- Bei Klassen mit Konstruktoren kann der Klassenname als Methode gedacht werden, die die Wandlung durchführt.
- In der Klasse *complexT* definiert der Konstruktor

`complexT (double re_in);`

die Umwandlung einer Fließkommazahl in eine komplexe Zahl.

- Der Konstruktor

`complexT (double re_in, double im_in);`

definiert die Umwandlung von zwei Fließkommazahlen in eine komplexe Zahl.

Typumwandlungen

Prog2_Typumwandlung1.cpp

// Explizite Typwandlung für complexT mittels Konstruktor

```
class complexT {  
    double re, im;  
  
public:  
    complexT (void)          { re = im = 0.0; }  
  
    complexT (double re_in)  { re = re_in; im =0.0; }  
  
    complexT (double re_in, double im_in)  
        { re = re_in; im = im_in; }  
};
```


Typumwandlungen

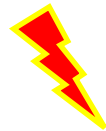
```
// main
```

```
int main () {
```

```
    complexT c1, c2;           // 2x Aufruf Konstr. #1  
    c1 = complexT (10);        // Aufruf Konstr. #2  
    c2 = complexT (10,11);     // Aufruf Konstr. #3  
    :
```

- Kann die Typwandlung eindeutig durchgeführt werden, kann wie in Standard-C die Angabe des Wandlungs-(Cast-)operators fehlen.

```
    c1 = 10;  
    c2 = (10, 11);             // ??????????
```



=> "left-hand operand of comma has no effect"

Typumwandlungen

- Die Konvertierung mittels Konstruktor wird auch bei der Parameterübergabe durchgeführt:

```
// IsZero
```

```
int IsZero (complexT c, double Epsilon = 0.000000001) {  
    return ((abs(c.re) <= Epsilon) && (abs(c.im) <= Epsilon));  
}
```

```
// main
```

```
int main () {  
    int k = 0;  
    int i = IsZero (k); // Konstruktor 2 nach Umwandlung  
}                       // int -> double durch Compiler
```

Typumwandlungen

- Verwendung eines expliziten Konstruktoraufrufs zur Initialisierung:

```
complexT c[2] = { complexT (1.0, 2.0),  
                 complexT (3.0, 4.0) };
```

- Auf die explizite Angabe des Konstruktors kann nicht verzichtet werden, die Initialisierungen von c1 und c2 führen *nicht* zum gewünschten Ergebnis:

```
complexT c1[2] = ((1.0, 2.0), (3.0, 4.0));  
complexT c2[4] = (1.0, 2.0, 3.0, 4.0);
```



Typumwandlungen

- In C++ ist es möglich, durch einen speziell überladenen Konstruktor eine Klasse in eine beliebige **andere Klasse** zu wandeln:

// Umwandlung vom Typ complexT in TestT

```
class TestT {  
    double f;  
public:  
    TestT (void);  
    TestT (double f_in);  
    TestT (complexT c_in);  
  
    /* ..... weitere Mitglieder von TestT ..... */  
};
```

Nachteil?:

TestT muss Zugriff auf die Attributwerte von complexT haben

Explizite Typumwandlungen (Casts)

- Eine Typwandlung vom Typ des Objekts in Richtung eines einfachen Datentyps, ist durch spezielle Klassenfunktionen ("Operator-Methoden") möglich:

```
// Definition der Operatorfunktion double
```

```
class complexT {
```

```
    double re, im;
```

```
public:
```

```
    /* ..... Weitere Mitglieder ..... */
```

```
    // Der operator double wandelt die Instanz in den
```

```
    // Typ double
```

```
    operator double (void) { return re; }
```

```
};
```

Typumwandlungen

- `complexT` kann mit einem Operator versehen werden, der die eigene Instanz in den Typ `TestT` wandelt:

```
// Umwandlung vom Typ complexT in TestT
```

```
class TestT;    // Prototyp falls TestT noch nicht definiert ist
```

```
class complexT {  
    double re, im;
```

```
public:
```

```
    /* ... Konstruktoren von complexT ... */
```

```
    // Der Operator TestT wandelt eine Instanz von  
    // complexT in eine Instanz von TestT.
```

```
    operator TestT ( void ) { return TestT ( re + im); }  
};
```

- *operator TestT* verwendet einen *TestT*-Konstruktor (Nummer 2) um ein *TestT*-Objekt zu erzeugen.

Vererben von Operatormethoden

- Operatormethoden können wie jede "normale" Funktion einer Klasse an die Ableitungen vererbt werden.
- Ausnahme: Operatormethoden für Zuweisungen werden nicht vererbt.
- Der Compiler erzeugt für eine Klasse einen Standard-Zuweisungsoperator, wenn der Programmierer keinen Zuweisungsoperator definiert hat.

(Prog2_vererb_opzuweis1.cpp)

(Prog2_vererb_opzuweis2.cpp)

Vererben von Operatormethoden

```
class AT {  
    int i;  
  
public:  
    AT (int i_in = 7) { i = i_in; }  
  
    AT &operator = ( const AT &arg_i ) {  
        cout << "AT Operator = aufgerufen" << endl;  
        i = arg_i.i;  
        return *this;  
    }  
  
    void print() { cout << "i = " << i << endl; }  
};
```


Vererben von Operatormethoden

```
class BT : public AT {
    int k;
public:
    BT (int i_in=5, int k_in=42) : AT(i_in) { k = k_in; }

    void print() {AT::print(); cout << "k = " << k <<endl;}
};

int main () {
    BT b1(1,2),b2(3,4);

    b2 = b1;
    b2.print(); // Was wird gedruckt ?
}
```

Vererben von Operatormethoden

// Ableitung definiert einen Zuweisungsoperator

```
class BT : public AT {
    int k;
public:
    BT (int i_in=5, int k_in=42) : AT(i_in) { k = k_in; }

    BT &operator = (const BT &arg_k) {
        cout << "BT Operator = aufgerufen" << endl;

        // Zuweisungsoperator AT für i aufrufen!
        AT::operator = ((AT)arg_k);

        k = arg_k.k;    // k selbst bearbeiten

        return *this;
    }

    void print() {AT::print(); cout << "k = " << k << endl;}
};
```

Vererbung / Variablentypen / Polymorphismus

- C++ ermöglicht eine Verlagerung der Typprüfung vom Übersetzungszeitpunkt zum Ausführungszeitpunkt eines Programms.
- Der Unterschied zu den bekannten *unions* besteht darin, daß der augenblickliche Typ des Wertes (z. B. *int* oder *float*) automatisch mitgespeichert wird.
- Bei der Übersetzung eines Methodenaufrufs kann vom Compiler nun nicht mehr eine feste Adresse eingesetzt werden, da die Adresse der Methode evtl. zur Laufzeit des Programms vom momentanen Typ des Objekts abhängig ist (late binding).
- Programmtechnisch umgesetzt wird der Polymorphismus mittels Zeigern auf Objekte der Basisklassen und virtuelle Methoden in der Klassenhierarchie.

Zuweisungskompatibilität in Klassenhierarchien

- In Klassenhierarchien gilt, dass Objekte von Ableitungen einer Klasse auch an Objekte der Basisklasse zugewiesen werden können, nicht aber umgekehrt.

```
class AT {  
    int i, j, k;  
};
```

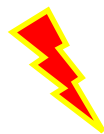
```
class BT : public AT {  
    double x, y;  
};
```

- Die Zuweisung

```
AT a; BT b; a = b;
```

ist zulässig, der umgekehrte Fall

```
b = a;
```



dagegen nicht.

Zuweisungskompatibilität in Klassenhierarchien

- Ein Zeiger vom Typ einer Klasse kann auch auf Instanzen aller abgeleiteten Klassen, nicht aber auf Instanzen von Basisklassen zeigen.

```
AT *ap = new AT;    // Basisklasse
```

```
BT *bp = new BT;    // Ableitung
```

```
ap = bp;
```

```
bp = ap;    // geht nicht !
```



Virtuelle Methoden / Polymorphismus

(vormals Teile 15-16)

- In der Ableitungskette können überschriebene Mitgliedsfunktionen (Methoden) je nach aktuellem Objekttyp dynamisch angesprochen werden:

(Prog2_virtual1.cpp)

(Prog2_virtual2.cpp)

Virtuelle Methoden

```
class AT {
public:

    void Doit() { cout << "AT::Doit" << endl; }
};

class BT : public AT {
public:

    void Doit() { cout << "BT::Doit" << endl; }
};

int main() {
    AT * ap;
    BT * bp = new BT;
    ap = bp;
    ap->Doit();
    delete bp;
}
```

Was ist mit der delete-Anweisung ?

Virtuelle Methoden

```
class AT {  
public:  
  
    virtual void Doit() { cout << "AT::Doit" << endl; }  
};  
  
class BT : public AT {  
public:  
  
    virtual void Doit() { cout << "BT::Doit" << endl; }  
};  
  
int main() {  
    AT * ap;  
    BT * bp = new BT;  
    ap = bp;  
    ap->Doit();  
    delete bp;  
}
```


Virtuelle Methoden

- Die erweiterte Zuweisungskompatibilität in Klassenhierarchien ermöglicht im letzten Programm die Zuweisung $ap = bp$. Die umgekehrte Zuweisung ist nicht ohne weiteres möglich (explizite Typwandlung erforderlich).
- Eine virtuelle Methode wird normal vererbt. Soll sie in einer Ableitung redefiniert werden, muß sie mit identischer Parameterliste deklariert werden. Das Schlüsselwort *virtual* kann dann fehlen, es wird automatisch ergänzt.
- Eine nicht-virtuelle Methode kann in einer Ableitung virtuell deklariert werden. Der umgekehrte Weg ist nicht möglich.

Virtuelle Methoden (override)

- Ergänzung: ab C++11 kann mit dem Schlüsselwort „override“ dem Compiler mitgeteilt werden, dass hier eine Überschreibung explizit gewollt ist. Stimmen die Signaturen nicht exakt überein, kann der Compiler einen Fehler auslösen und die Übersetzung abbrechen, ansonsten würde er eine Überladung anlegen. Beispiel:

```
class AT {  
public:  
  
    virtual void Doit() { cout << "AT::Doit" << endl; }  
};  
  
class BT : public AT {  
public:  
  
    virtual void Doit() override { cout << "BT::Doit" << endl; }  
};
```

Virtuelle Methoden -- Late Binding

- Ist eine Methode virtuell definiert, wird die Zuordnung zwischen Methodenaufruf und aufgerufener (Überschreibung der) Methode erst zur Laufzeit des Programms hergestellt (late binding).
- Bei late binding wird ein Sprung zu einer generellen Verteilerfunktion codiert, die eine Tabelle (*vtbl*) mit Adressen in Frage kommender Methoden erhält.

Abstrakte Klassen / Methoden

- Klasse mit *semi-abstrakter* Methode:

```
class AT {  
    int i;  
public:  
    virtual void f() {}  
};
```

- Vergisst man in einer Ableitung `f` zu redefinieren, wird natürlich **AT::f** vererbt, und bei einem Aufruf von `f ()` passiert nichts.

Abstrakte Klassen/Methoden

- Ersetzt man
durch
handelt es sich um eine echte abstrakte Methode (rein virtuelle Methode).

```
virtual void f() {}  
virtual void f() = 0;
```
- Eine Ableitung von AT muß nun explizit entweder eine *nicht-abstrakte* oder wiederum **eine abstrakte Methode `f()`** implementieren.
- Das **einfache Erben von `f()` ist nicht mehr möglich.**
- Eine **Methode ohne Funktionalität nennt man auch *abstrakte Methode* oder *rein virtuelle Methode*.** Eine Klasse mit einer oder mehreren abstrakten Methoden heißt auch *abstrakte Klasse*.

Virtuelle Methoden und Konstruktoren

- Die *vtbl* wird durch den Konstruktor einer Klasse mit virtuellen Methoden erzeugt bzw. erweitert.
Dies geschieht *nach* der Abarbeitung aller Basisklassenkonstruktoren, aber *vor* dem Betreten des Anweisungsblocks des Konstruktors.
- Grundsätzlich sollte man den Aufruf virtueller Methoden in Konstruktoren vermeiden.
- Der Standardkonstruktor sollte einen Ausgangszustand des Objekts herstellen, der auch ohne Aufruf virtueller Methoden auskommt.

Polymorphismus

- Virtuelle Methoden und das late binding machen das polymorphe Verhalten der Objekte einer Klassenhierarchie möglich.
- *Programmbeispiel Fisch / Haifisch / Seepferdchen*
(*Prog2_HaifischSeepferdchen.cpp*)

live Demo

- Die Klassen in der Hierarchie bilden eine *Gruppe* deren Sinn klar wird, wenn man *Operationen auf der Gruppe* ausführt.

9. Vorlesung

try – catch – throw

- macht Ereignissteuerung im klassischen C++ möglich
- optimiert Fehlerbearbeitung
- Code / Syntax:

```
try {  
    // beliebige Anweisungssequenz  
}  
catch (char * errstr) {  
    cout << "catching char * error: " << errstr << endl;  
    return 22; // oder Fehlerbearbeitung  
}  
catch(...) {    // 'Lumpensammler'-catcher  
    cout << "an unknown error occurred" << endl;  
    return 2;   // oder Fehlerbearbeitung  
}
```

... in einer Methode:

```
if (error) {  
    throw ("an error happened");  
}
```

Templates (generische Programmierung)

- Templates sind „Vorlagen“ für Klassen oder Funktionen
- Der genaue Typ ist in der Template-Definition offen gelassen (generisch)
- Er wird bei der Verwendung spezifiziert
- Die „Parameter“ der Templates werden in spitzen Klammern < ...> angegeben
- Parameter der Templates sind typischerweise Klassen, können aber auch andere Datentypen oder Instanziierungs-Parameter sein

Beispiel Funktions-Template

```
// Template-Definition
```

```
template <class Flexible> void swapmeall (Flexible & obj1,  
                                         Flexible & obj2) { // Funktions-Template  
    Flexible helpme;  
    helpme = obj2;  
    obj2 = obj1;  
    obj1 = helpme;  
}
```

```
// Anwendungsbeispiel
```

```
class cT1 {                                // einfache Klasse  
    double x;  
public:  
    cT1 (double x1 = 0.0) { x = x1; }  
  
    void print (void) {  
        cout << "cT1-Objekt Variable x: " << x << endl;  
    }  
};
```

Beispiel Funktions-Template

```
// Hauptprogramm mit Anwendung Template bei diversen Typen
int main() {
    int a = 11, b = 22;
    double t = 3.333, u = 4.444;
    cT1 v(56.8), w(67.9);

    // Int Variablen tauschen
    cout << "a = " << a << endl;
    swapmeall (a, b);
    cout << "a = " << a << endl;

    // Double Variablen tauschen
    cout << "t = " << t << endl;
    swapmeall (t, u);
    cout << "t = " << t << endl;

    // Klassen-Objekte tauschen
    v.print();
    swapmeall(v, w);
    v.print();

    return 0;
}
```

Beispiel Klassen-Template

```
// Template-Definition
```

```
template<class T, int i>
class A // A ist das Klassen-Template
{
    T * ct;
public:
    A() { // oder A<T, i>::A<T, i>()
        ct = new T[i]; // Array der Laenge i aus Objekten T
    }

    void hello() { cout << "hello" << endl; }
};
```

```
// Anwendungsbeispiel
```

```
class cT1 { // einfache Klasse
    double x;
public:
    cT1 (double x1 = 0.0) { x = x1; }

    void print (void) {
        cout << "cT1-Objekt Variable x: " << x << endl;
    }
};
```

Beispiel Klassen-Template

// Hauptprogramm mit Klasseninstanzen diversen Typen

```
int main() {  
    A<int, 30> integervector; // integervector ist die Instanz  
                                // von A für ein Integer-Array  
    A<cT1, 10> t1vector;      // Array aus T1-Objekten  
  
    // Instanzen verwenden  
  
    t1vector.hello();  
    integervector.hello();  
  
    return 0;  
}
```

Templates (generische Programmierung)

Anwendungen für Templates (vorgefertigte APIs in Bibliotheken):

- Zeiger: smart / shared pointers
 - eigene Klasse, gekapselte Objekte, Schnittstellen-Methoden
 - Attribut ist ein „echter“ Zeiger auf den Template-Typ
 - Zeiger werden gesetzt über überladene Zuweisungsoperatoren
 - Dereferenzierung über überladene Zugriffsoperatoren
 - bei shared pointers: Mitzählen, wie viele Zeiger auf ein Objekt zeigen; wenn keiner mehr darauf zeigt, Objekt löschen
- Listen (generische verkettete Listen)
 - eigene Klasse, gekapselte Objekte
 - Attribute: Nutz-Objekt vom Template-Typ, Zeiger für doppelt verkettete Liste
 - Schnittstellen-Methoden zum Einfügen, Finden, Löschen von Elementen an beliebigen Stellen der Liste, Operationen für die gesamte Liste (Kette)
- weitere generische Konstruktionen ...

Streams (Vertiefung / Implementierungsdetails)

(vormals Teil06)

- In C stehen zur Ein- Ausgabe von Daten Funktionen wie *printf* und *scanf* zur Verfügung. C++ beinhaltet zusätzlich sogenannte *Streams*.

```
//    Hello World mit Streams
#include <iostream>
using namespace std;

int main()  {

    cout  <<  "good morning, world  .... ";

}
```

- *cout* ist der Standard-Ausgabestrom, vergleichbar mit *stdout* in C

Streams (Vertiefung / Implementierungsdetails)

```
//    Eingabe eines integers mit Streams
#include <iostream>
using namespace std;

int main()    {

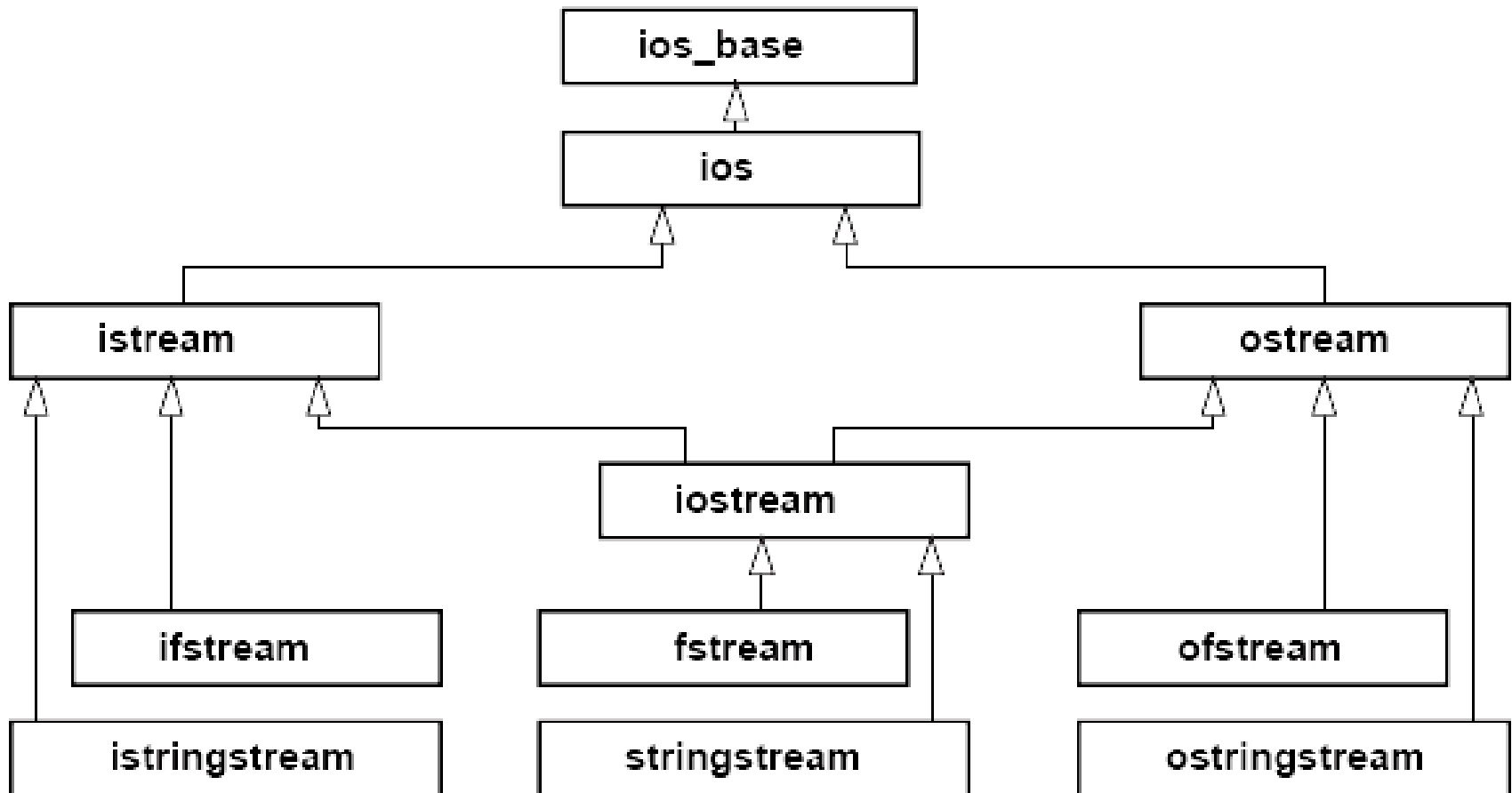
    cout  <<  "Bitte einen Wert für i eingeben : „

    int i;
    cin  >>  i;

}
```

- *cin* ist der Standard-Eingabestrom, der normalerweise mit der Tastatur verbunden ist (stdin).

Streams Klassenhierarchie



aus: Breymann C++ Hanser 2007

Streams sind Objekte

- *cout* und *cin* sind Objekte von Klassen für die die Operatoren << bzw. >> geeignet überladen wurden.
- Definition der << und >> Operatoren als „Mitglieds-“ funktionen der Streamklassen *istream* und *ostream* (Beispiele) :

```
istream& operator >> ( istream&, int& );  
ostream& operator << ( ostream&, const char* );
```

- Die Stream-Library ist ein Bestandteil der Standardbibliothek des Compilers.

Stream-Library

- Zur Arbeit mit Streams sind die folgenden Headerdateien wichtig:

<code>iostream.h</code>	Grundlegende Deklarationen
<code>iomanip.h</code>	Deklarationen für Manipulatoren
<code>fstream.h</code>	Deklarationen zur E/A mit Dateien
<code>strstream.h</code>	Deklarationen zur E/A mit Strings

- Auszug aus *iostream.h*:

```
namespace std {           // Auszug aus <iostream>
    extern istream cin;    // Standardeingabe
    extern ostream cout;  // Standardausgabe
    extern ostream cerr;  // Standardfehlerausgabe
    extern ostream clog;  // gepufferte Standardfehlerausgabe
}
```

Stream-Library

- Standard-Streams die durch Includieren der Headerdatei *iostream.h* verfügbar werden:

<code>cin</code>	Standard-Eingabe (Tastatur)
<code>cout</code>	Standard-Ausgabe (Bildschirm)
<code>cerr</code>	Standard-Fehlerausgabe (Bildschirm)
<code>clog</code>	Standard-Protokoll (Bildschirm)

Prog2_tripleout.cpp

- Die Objekte werden in *iostream.h* definiert und in der Stream-Library mit Tastatur und Bildschirm verbunden.
- Die Standard-Streams können ohne weitere Deklaration oder Initialisierung sofort verwendet werden.

Operatoren << und >>

- Die Operatoren << und >> sind mehrfach überladen und führen die eigentliche Ausgabe bzw. Eingabe für die unterschiedlichen Datentypen aus:

```
istream& operator >> ( istream& is, int&);  
:  
ostream& operator << ( ostream& os, const char* ch );  
ostream& operator << ( ostream& os, char );  
ostream& operator << ( ostream& os, double );  
:
```

- In *iostream.h* sind die Übergabeoperatoren für die grundlegenden Datentypen *char*, *short*, *int*, *long*, *float*, *double*, *long double* sowie sinnvolle Kombinationen mit *unsigned*, *** und *&* sowie für *Zeiger* bereits als *ostream* und *istream* Definitionen vorhanden.
- Für eigene Datentypen können die Operatoren überladen werden.

Prog2_ueberlad_istream_op.cpp

Ausgabe unterschiedlicher Datentypen

```
//-- Ausgabe unterschiedlicher Datentypen mit <<
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main () {
```

```
    char* s = "Ein String";
```

```
    int i = 1;
```

```
    float f = 3.14159;
```

```
    void* p = &i;
```

```
    cout << '\n'; cout << "string   :"; cout << s;
```

```
    cout << '\n'; cout << "integer  :"; cout << i;
```

```
    cout << '\n'; cout << "float    :"; cout << f;
```

```
    cout << '\n'; cout << "Zeiger   :"; cout << p;
```

```
}
```

- Eine wichtige Eigenschaft der Übergabeoperatoren ist, daß sie eine Referenz auf den eigenen Stream zurückgeben (die sie auch als Parameter erhalten).

Kaskadierung

- Dadurch können die Operatoren kaskadiert werden:

```
int main ()    {  
  
    char* s = "Ein String";  
    int    i = 1;  
    float  f = 3.1415;  
    void*  p = &i;  
  
    cout << '\n' << "string      :" << s  
         << '\n' << "integer    :" << i  
         << '\n' << "float      :" << f  
         << '\n' << "Zeiger    :" << p;  
  
}
```

- Die Priorität eines Operators wird beim Überladen nicht geändert.
- In Kaskaden werden die Übergabeoperatoren von links nach rechts ausgewertet.

Formatierungen

- Mit Streams sind die gleichen Formatierungen wie in Standard-C möglich:
 - Feldbreite
 - Bündigkeit (rechts, links)
 - Füllzeichen
 - Zahlenformat, Vorzeichen, Nachkommastellen
- Streams ermöglichen das Ignorieren bzw. Beachten von Leerzeichen (whitespace) bei der Eingabe.
- Einige Formatangaben werden nach jeder Ausgabe wieder auf einen Standardwert zurückgesetzt.

Formatierungen

- Die für eine E/A Operation zu verwendenden Formatierungen werden als Variablen bzw. als einzelne Bits im Stream gespeichert.
- Die Variablen bzw. Bits müssen vor einer Ausgabe entsprechend gesetzt werden.

```
int  x_width;    //  Feldbreite für die Ausgabe
int  x_fill;     //  Füllzeichen für die Ausgabe
int  x_precision; //  Genauigkeit für Fließkomma
long x_flags;    //  einzelne Bits
```

Formatierungen

- Für `x_flags` ist in der Klasse `ios` ein Aufzählungstyp:

```
enum {
    skipws      = 0x0001,    // skip whitespace (in)
    left        = 0x0002,    // left-adjust (out)
    right       = 0x0004,    // right-adjust (out)
    internal    = 0x0008,    // padding (sign or base indicator)
    dec         = 0x0010,    // decimal conversion
    oct         = 0x0020,    // octal conversion
    hex         = 0x0040,    // hexadecimal conver.
    showbase    = 0x0080,    // use base indicator
    showpoint   = 0x0100,    // force decimal point
    uppercase   = 0x0200,    // upper-case hex (out)
    showpos     = 0x0400,    // add '+' to integers
    scientific  = 0x0800,    // use 1.2345E2 floating notation
    fixed       = 0x1000,    // use 123.45 float. not.
    unitbuf     = 0x2000,    // flush all streams after insertion
    stdio       = 0x4000,    // flush stdout, stderr after insertion
};
```

Formatierung durch Mitgliedsfunktionen

- *istream* und *ostream* enthalten eine Reihe von Mitgliedsfunktionen, über die Formate angegeben und abgefragt werden können.

```
// Mitgliedsfunktionen für die Formatangabe
```

```
// reading/setting field width
```

```
int width();  
int width( int );
```

```
// reading/setting padding character
```

```
char fill();  
char fill( char );
```

```
// reading/setting digits of floating precision
```

```
int precision();  
int precision( int );
```

Formatierung durch Mitgliedsfunktionen

```
// Ausgabe eines Betrags mit Hilfe von Füll-  
//   zeichen in einem Feld der Breite 10  
  
// ...  
  
int amount = 355;  
  
cout << '\n' << "Sie erhalten heute EUR ";  
cout.width( 10 );  
cout.fill( '*' );  
cout << amount << '\n';  
  
// ...
```

Formatierung durch Mitgliedsfunktionen

- Für die in x_flags codierten Formate gibt es die Mitgliedsfunktionen

```
long flags();
```

```
long flags( long );
```

```
long setf( long _setbits, long _field );
```

```
long setf( long );
```

```
long unsetf( long );
```

Formatierung durch Mitgliedsfunktionen

Prog2_cout_mfunk.cpp

```
// Ausgabe einer Umrechnungstabelle
```

```
cout << '\n';
cout.width( 12 ); cout << "dezimal";
cout.width( 12 ); cout << "oktal";
cout.width( 12 ); cout << "hexadezimal";

for (int i = 0; i <= 16; i++) {
    cout << '\n';
    cout.width(12); cout.unsetf(ios::oct | ios::hex);
    cout.setf(ios::dec); cout << i;
    cout.width(12); cout.unsetf(ios::dec | ios::hex);
    cout.setf(ios::oct); cout << i;
    cout.width(12); cout.unsetf(ios::dec | ios::oct);
    cout.setf(ios::hex); cout << i;
}
cout << '\n';
```

Formatierung durch Manipulatoren

- Ein *Manipulator* ist eine spezielle Mitgliedsfunktion die Formatierungsdaten im Stream ändert, aber wie eine Variable in Ausgabeanweisungen erscheint.

// Verwendung der Manipulatoren setw und setfill

```
#include <iostream>
#include <iomanip>
using namespace std;
```

```
int main() {
```

```
    // Ausgabe eines Betrags mit Hilfe von Füll-
    // zeichen in einem Feld der Breite 10
```

```
    int amount = 355;
    cout << '\n' << "Sie erhalten heute EUR "
         << setw( 10 ) << setfill( '*' )
         << amount << '\n';
```

```
}
```


Formatierung durch Manipulatoren

- Definierte *Manipulatoren*:

`ws` whitespace bei der Eingabe ignorieren

`dec` Zahl in dezimaler Form ausgeben

`oct` Zahl in oktaler Form ausgeben

`hex` Zahl in hexadezimaler Form ausgeben

`setbase(int)` Basis (0, 8, 10 oder 16) setzen (0: zurück auf 10)

`setfill(int)` Füllzeichen setzen

`setw(int)` Feldbreite setzen

`setprecision(int)` Genauigkeit für Fließkomma

`setiosflag(long)` entspricht `setf(long)`

`resetiosflag(long)` entspricht `unsetf(long)`

Formatierung durch Manipulatoren

Prog2_cout_manip.cpp

```
// Ausgabe einer Umrechnungstabelle
```

```
cout << '\n' << setw( 12 ) << "dezimal"  
      << setw( 12 ) << "oktal"  
      << setw( 12 ) << "hexadezimal";  
  
for (int i = 0; i <= 16; i++) {  
    cout << '\n' << setw(12) << dec << i  
          << setw(12) << oct << i  
          << setw(12) << hex << i;  
}
```

- Die Deklaratoren für *ws*, *dec*, *oct* und *hex* befinden sich in *iostream.h*, die für die restlichen Manipulatoren in *iomanip.h*.

Formatierung durch Manipulatoren

- Weitere Manipulatoren:

<code>endl</code>	Neue Zeile beginnen
<code>ends</code>	Stringendezeichen anhängen
<code>flush</code>	Ausgabe-/Eingabepuffer leeren

```
// Verwendung des Manipulators endl

// Ausgabe eines Betrags mit Hilfe von Füll-
// zeichen in einem Feld der Breite 10

int amount = 355;

cout << endl << "Sie erhalten heute EUR „
      << setw( 10 ) << setfill( '*' )
      << amount << endl;
```

Fin!

Bernd Ruhland

email: ruhland@hs-worms.de