Fachhochschule Köln
Cologne University of Applied Sciences

Softwaretechnik

Fomuso Ekellem

WS 2011/12

Weiteren Verlauf der Vorlesung

- **12.12.2011(2 Std) Implementationsphase**
- **19.12.2011(2 Std) Test-, Abnahme-, Einführungs-, Wartung- und Pflegephase**

- **16.01.2012(2 Std) Produkte und Recht , Vorlesung-Zusammenfassung**
- **23.01.2012(4 Std) Projektvorführung und Dokumentationsvorführung**
- **30.01.2012(4 Std) Test und Besprechung- Noten Vergabe**

Noten

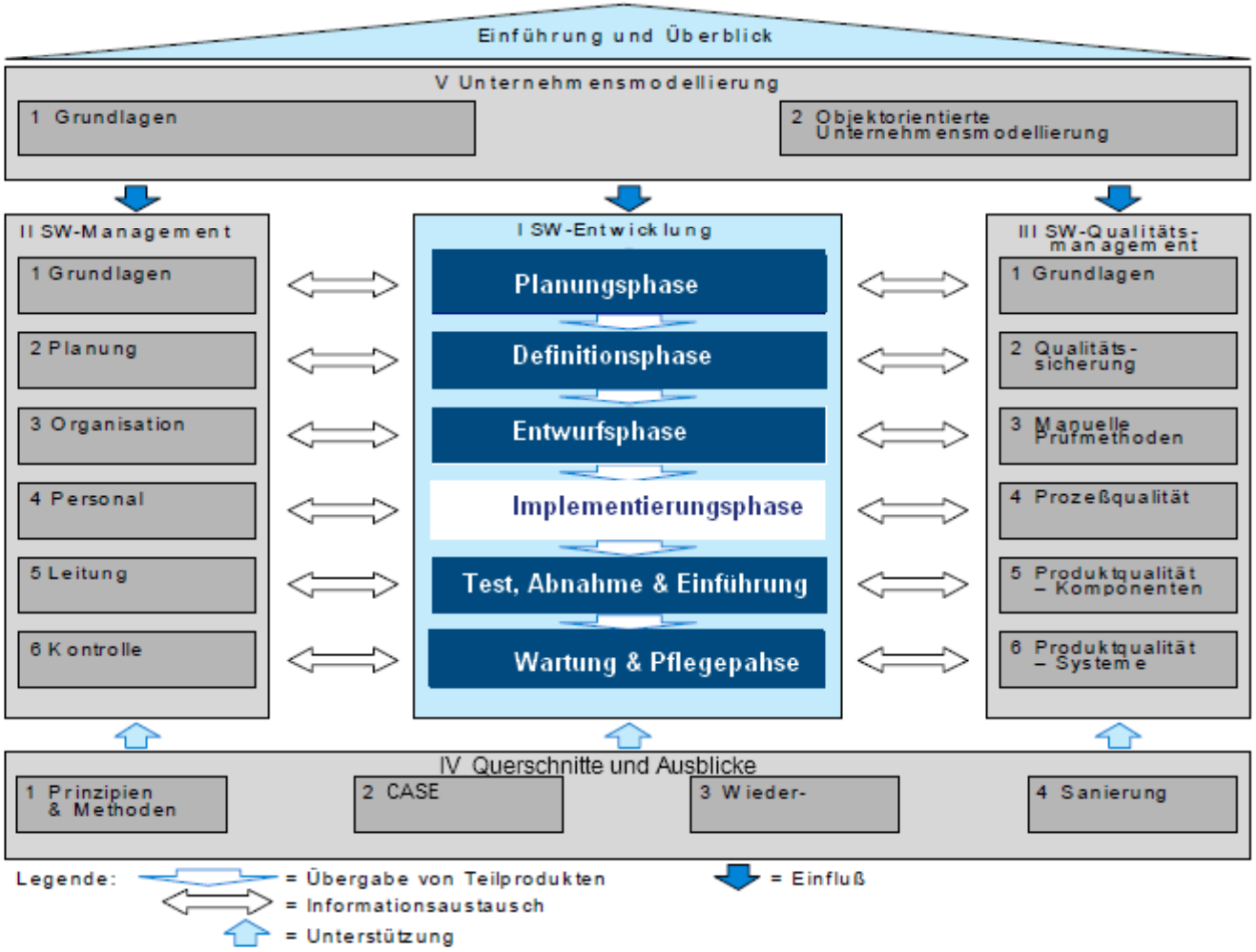
- **50% Projektarbeit**
- **50% Test**
  - **Test mit 25Punkte: Sie brauchen 13 Punkte zu bestehen. 10 Punkte können Sie von der Übungen sammeln, dann brauchen Sie nur noch 3 Punkte zu bestehen.**
  - **4 Fragen aus der Phasen(Definitionsphase, Entwurfsphase, Implementierung, Test) und 1 Frage aus Rechte.**

Inhalt

- Implementierungsphase
  - Ziele
  - Grundprinzipien
  - Programmiersprachen
- Implementierung mit C++
  - C und C++
  - Grundlage der Programmierung
    - Operatoren (Arithmetische, Verhältnis, Logische, Zeichen, Bit, Shift und Zuordnen)
    - Ausdrücke,
    - Schleifen
    - Arrays
    - Zeiger, Referenzen und dynamische Speicher-Allokation
  - OOP Basis
  - OOP Specials

Implementierungsphase

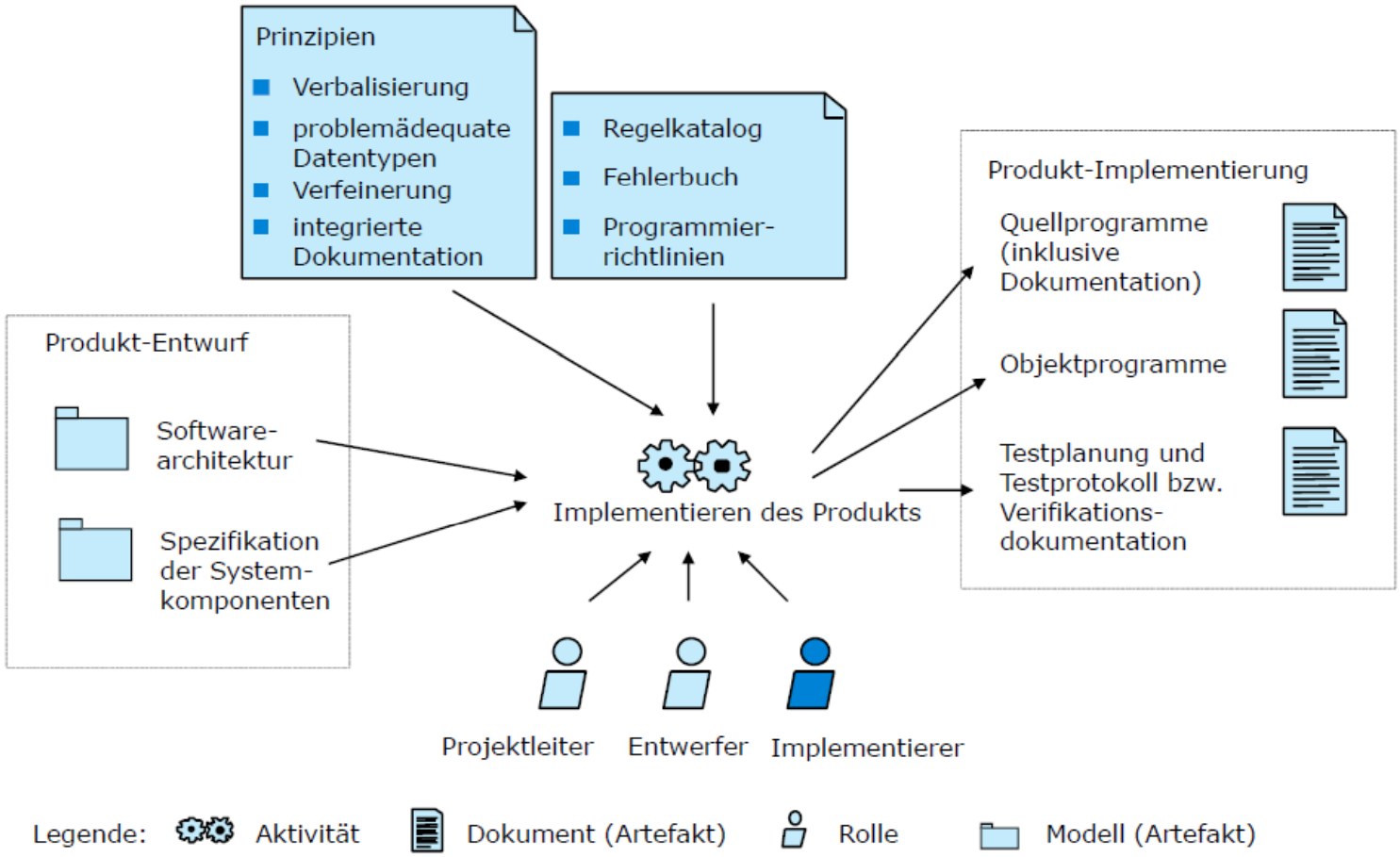
In der Implementierungsphase findet die eigentliche Programmierung der Software statt.



Implementierungsphase

- **Lernziele**
  - Wissen, was die Ziele der Implementierung sind
  - Verstehen, was eine gute strukturierte Programmierung ausmacht
  - Erklären können, was die Merkmale eines guten Programmierstils sind
- **Ziel der Implementierung**
  - Ausgangspunkt: Entwurfsspezifikation (Systemarchitektur)
  - Transformation der Entwurfsergebnisse in Programme, die auf einem bestimmten Zielrechner ausführbar sind
  - Ergebnis: Programme

Implementierungsphase



Implementierungsphase

- **Aktivitäten**
  - Konzeption von Datenstrukturen und Algorithmen
  - Strukturierung des Programms durch geeignete Verfeinerungsebenen
  - Dokumentation der Problemlösung und der Implementierungsentscheidungen
  - Umsetzung der Konzepte in die Konstrukte der verwendeten Programmiersprache
  - Angaben zur Zeit- und Speicherkomplexität
  - Test oder Verifikation des Programms einschl. Testplanung und Testfallerstellung
- Auch »Programmieren im Kleinen« genannt.

Implementierungsphase

- **Teilprodukte**
  - Quellprogramm einschl. integrierter Dokumentation
  - Objektprogramm
  - Testplanung und Testprotokoll bzw. Verifikationsdokumentation
- Alle Teilprodukte aller Systemkomponenten müssen integriert und einem Systemtest unterzogen werden
- **Rollen**
  - Implementierer / Programmierer / Algorithmenkonstrukteur.
- Basiskonzepte zur Implementierung der Systemkomponenten:
  - Kontrollstrukturen
  - Entscheidungstabellen
- Bei der Implementierung sollten bestimmte Prinzipien eingehalten werden

Grundprinzipien

- **Verbalisierung: Gute Verbalisierung**
  - Aussagekräftige, mnemonische Namensgebung
  - Geeignete Kommentare
  - Selbstdokumentierende Programmiersprache

  Vorteile der Verbalisierung
  - Leichte Einarbeitung in fremde Programme bzw. Wiedereinarbeitung in eigene Programme
  - Erleichtert »code reviews«, Modifikationen und Wartung
  - Verbesserte Lesbarkeit der Programme

Grundprinzipien-Datentypen

- Daten- und Kontrollstrukturen eines Problems sollen sich in der programmiersprachlichen Lösung möglichst unverfälscht widerspiegeln.
- Programmiersprache sollte folgende Eigenschaften bezogen auf Datenstrukturen besitzen:
  - Umfangreiches Repertoire an Basistypen;
  - Geeignete Typkonstruktoren;
  - Benutzerdefinierbare Typen.
- Aufgabe des Programmierers: Angebot an Konzepten einer Programmiersprache optimal zur problemnahen Lösungsformulierung verwenden.

Grundprinzipien-Datentypen

- Regeln
  - Können die Daten durch Basistypen beschrieben werden, dann ist der geeignete Basistyp auszuwählen.
  - Der Wertebereich sollte so festgelegt werden, dass er möglichst genau das Problem widerspiegelt – unter Umständen durch Einschränkungen des Basistyps.
- Bei OO-Entwicklung
  - Typen von Attributen durch elementare Klassen modellieren, gilt auch für Aufzählungstypen
- Vorteile problemadäquater Datentypen
  - Verständliche, leicht lesbare, selbstdokumentierende und wartbare Programme
  - Statische und dynamische Typprüfungen verbessern die Qualität des jeweiligen Programms
  - Die Daten des Problems werden 1:1 in Datentypen des Programms abgebildet, d. h. Wertebereiche werden weder über- noch unterspezifiziert.

Grundprinzipien-Verfeinerung

- Dient dazu, ein Programm durch Abstraktionsebenen zu strukturieren.
- Verfeinerungsstruktur kann auf 2 Arten im Quellprogramm sichtbar gemacht werden.
  - Die oberste Verfeinerungsebene – bestehend aus abstrakten Daten und abstrakten Anweisungen – ist kompakt beschrieben.
    - Die Realisierung jeder Verfeinerung wird an anderer Stelle beschrieben.
  - Alle Verfeinerungsebenen sind substituiert
    - Die übergeordneten Verfeinerungen werden als Kommentare gekennzeichnet.
- Vorteile
  - Entwicklungsprozess im Quellprogramm dokumentiert
  - Leichtere und schnellere Einarbeitung in ein Programm
  - Entwicklungsentscheidungen können besser nachvollzogen werden

Grundprinzipien-Dokumentation

- Integraler Bestandteil jedes Programms
- Angaben
  - Kurzbeschreibung des Programms
  - Verwaltungsinformationen
  - Kommentierung des Quellcodes
- Programmvorspann
  - Programmname: Name, der das Programm möglichst genau beschreibt
  - Aufgabe: Kurzgefasste Beschreibung des Programms einschl. der Angabe, ob ein GUI- , ein Fachkonzept- oder ein Datenhaltungs-Programm bzw. eine entsprechende Klasse (bei OOP) vorliegt.
  - Zeit- und Speicherkomplexität des Programms
  - Name der Programmautoren
  - Versionsnummer Datum (Release-Nummer, Level-Nummer)

Programmiersprachen

Qualitätskriterien für Programmiersprachen

- Datenstrukturen
  - Verfügbarkeit von Datenstrukturen als Sprachmittel
  - Datenstrukturen älterer Programme (Felder, Zeiger auf Datenstrukturen) sind gefährlich, da sie unbeschränkte Zugriffe ermöglichen und zur Laufzeit in Umfang und Struktur geändert werden können.
  - Sprache mit eigenen Elementen für Datenstrukturen sind sicherer und besser lesbar
  - erweiterbare abstrakte Datentypen in objektorientierten Programmiersprachen ermöglichen flexible und erweiterbare Lösungen
- Ablaufsteuerung
  - Ausnahme-und Unterbrechungsbehandlung in technischen Anwendungen
  - Parallele Prozesse und Synchronisierung

Programmiersprachen

Qualitätskriterien für Programmiersprachen weiter…

- Effizienz
  - C erlaubt sehr effiziente Programme, da maschinennahe Programmiert wird.
  - objektorientierte Sprachen erlauben weniger effiziente Programme
  - Effizienz kann durch guten optimierenden Compiler nachhaltig beeinflusst werden
- Sicherheit
  - Sicherheit einer Programmiersprache wird bestimmt durch Lesbarkeit und Mechanismen zur Typprüfung wie Templates in C++ oder generische Klassen in Java.
- Portabilität
  - Verfügbarkeit von Compilern auf verschiedenen Rechnern

Programmiersprache

Qualitätskriterien für Programmiersprachen weiter…

- **Dokumentationswert**
  - Voraussetzung für Lesbarkeit und Wartbarkeit .
  - Programme werden nur einmal geschrieben, aber oft gelesen
  - wichtig für langlebige Programme
  - explizite Schnittstellenbeschreibung (Java, C#)

Programmierstil

- Programme müssen verständlich sein
- Lesbarkeit eines Programms ist abhängig von
  - Programmiersprache
  - Programmierstil des Implementierers
- Elemente eines guten Programmierstils
  - Strukturiertheit
  - äußere Form
  - Ausdruckskraft
  - Effizienz

Programmierstil

Ausdruckkraft

- Namenwahl bei der Benennung von Objekten und Operationen
- aussagekräftige, konsistente Namen, auch wenn die Bezeichner lang werden.
- allgemein gebräuchliche Abkürzungen Schlechtes Bsp.:WPSMH = Wärmepumpensteuerung.
- keine Sprachmischung
- Groß-/Kleinschreibung Bsp.: große Anfangsbuchstaben für Datentypen, Klassen, Operationen; kleine Anfangsbuchstaben für Variablen
- Verwendung von Hauptwörtern für Werte, Zeitwörter für Tätigkeiten und Eigenschaftswörter für Bedingungen Bsp.: breite, readKey, gueltig

Programmierstil - Gebrauch von Kommentare

- Beschreibung des Wesentlichen kurz und präzise
  - jede Systemkomponente soll mit ausführlichem Kommentar beginnen
  - Was leistet die Komponente
  - Wozu wird die Komponente verwendet
  - Welche Verfahren, Prozeduren, ... werden benutzt
  - Wer ist der Verfasser
  - Wann wurde die Komponente geschrieben
  - Welche Änderungen wurden vorgenommen
- jede Prozedur soll mit Kommentar versehen werden, der die Aufgabe, Arbeitsweise und Schnittstellenbeschreibung erklärt
- Erläuterungen der Bedeutung von Variablen
- Erläuterung abgeschlossener Teilaufgaben (Programmblöcke)
- schwer verständliche Anweisungen, Nachführung der Kommentare bei Korrektur

Grundlage der Programmierung - Operatoren

- **Operatoren** führen **Aktionen** mit **Operanden** aus.
- Der **Zuweisungsoperator** <operand_A> = <operand_B> weist dem linken Operanden, welcher eine Variable sein muss, den Wert des rechten Operanden zu. Es können auch Mehrfachzuweisungen auftreten. Z.B. a= b= c= 200;
- **Arithmetische Operatoren**

```
x + y, x –Y          // Addition und Subtraktion
x * y, x / y         // Multiplikation und Division
x% y                 // Modulo (Rest bei ganzzahliger Division)
x++, ++x, y--, --y   // x++ entspricht x = x + 1; y-- entspricht y = y-1
+x, -y               // unitärer (Vorzeichen)- tritt nur ein Operand auf.
x++                  //nutze aktuellen Wert und Erhöhe um 1
++x                  //Erhöhe um 1 und nutze neuen Wert
```

Grundlage der Programmierung - Operatoren

- **Verhältnis Operatoren**

  Verhältnisse

  x < y // kleiner
  x <= y //Kleiner gleich
  x > y //größer
  x >= y //größer, gleich
  x == y //gleich?
  x != y //nicht gleich?

Beispiel:

```
{
        bool bi,bj;
        int i;
        bi = ( 3 <= 4 );
        bj = ( 3 > 4 );
        cout << " 3 <= 4 TRUE = " << bi << endl;
        cout << " 3 > 4 FALSE = " << bj << endl;
        // if - statement will be defined in Sec. 4
        i = 3;
        if (i <= 4)
        {
                cout << "\n i less or equal 4 \n\n";
        }
}
```

Grundlage der Programmierung - Operatoren

- **Logische Operatoren**

  Logik

  0 // false

  nicht null // true

  !X // Negation

  x **&&** y // logisch und

  x || y // logisch oder

  && und || werden von links nach rechts

  nach bedarf Ausgewertet

Beispiel:

```
{
        const int Ne = 5; // one limit
        int i;
        cout << " i = " ;
        cin >> i; // Input i
        if ( i <= Ne && i >= 0 ) // other limit is 0
        {
                cout << "i between 0 and 5" << endl;
        }
}
```

Grundlage der Programmierung - Operatoren

- **Zeichen Operatoren**

  Sonderzeichen

  '**n**' //neue Zeile

  '**"**' //Hochkomma

  '**""**' //Anführungszeichen

  '**?**' //Fragezeichen

Grundlage der Programmierung - Operatoren

- **Bit und Shift Operatoren**

  ~i // Komplement
  (bitweise Negation des Operanden)

  i&j // UND
  i^j // exklusives ODER
  i|j // inklusives ODER
  i<<n // schiebe nach links
  i>>n // schiebe nach rechts

  Nutzbar für alle integer Typen

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	L	0	L	L
L	0	0	L	L
L	L	L	L	0

Beispiel:

```
main()
{
short int k,l;
short int n1,n2,n3,n4,n5,n6,n7;
l = 5;          // 0..000101 = 5
k = 6;          // 0..000110 = 6
n1 = ~k;        // Komplement 1..111001 = -7 = -6 - 1
n2 = k & l;     // bit-AND 0..000100 = 4
n3 = k | l;     // bit-OR 0..000111 = 7
n4 = k ^ l;     // bit-XOR 0..000011 = 3
n5 = k << 2;    // shift left by 2 0..011000 = 24 = 6 * 2^2
n6 = k >> 1;    // shift right by 1 0..000011 = 3 = 6 / 2^1
n7 = l >> 1;    // shift right by 1 0..000010 = 2 = 5 / 2^1
}
```

Grundlage der Programmierung - Operatoren

- **Zuordnung (Inkrement- und Dekrement) Operatoren**

x = y

x += y, x -= y

X *= y, x/= y

x %= y

x >>= n, x <<= n

x &= y, x |= y

x ^= y

gewöhnungsbedürftig

ermöglicht kompakten Code

```
// Beispiel: prefix Notation
{
int i=3, j;
++i;            // i = 4
j = ++i;        // i = 5, j = 5
// prefix Notation oben entspricht
i = i + 1;
j = i;
}
```

```
// Beispiel: postfix Notation
{
int i=3, j;
i++;            // i = 4
j = i++;        // i = 5, j = 4
// postfix Notation oben entspricht
j = i;
i = i + 1;
}
```

```
{
int i,j,w;
float x,y;
i += j          // i = i+j
w >>= 1;        // w = w >> 1 (= w/2)
x *=y;          // x = x*y
}
```

Grundlage der Programmierung - Operatoren

- **Operationen mit vordefinierten Funktionen**

Funktion/Konstante	Beschreibung
`sqrt(x)`	Quadratwurzel von x: $\sqrt[2]{x}$ $(x \geq 0)$
`exp(x)`	e^x
`log(x)`	natürlicher Logarithmus von x: $\log_e x$ $(x > 0)$
`pow(x,y)`	Potenzieren ($x > 0$ falls y nicht ganzzahlig)
`fabs(x)`	Absolutbetrag von x: $\|x\|$
`fmod(x,y)`	realzahliger Rest von x/y $(y \neq 0)$
`ceil(x)`	nächste ganze Zahl $\geq x$
`floor(x)`	nächste ganze Zahl $\leq x$
`sin(x)`, `cos(x)`, `tan(x)`	trigonometrische Funktionen
`asin(x)`, `acos(x)`	trig. Umkehrfunktionen $(x \in [-1, 1])$
`atan(x)`	trig. Umkehrfunktion
`M_E`	Eulersche Zahl e
`M_PI`	π

Grundlage der Programmierung - Operatoren

- **Operationen mit vordefinierten Funktionen**

Funktion	Beschreibung
`strcat(s1,s2)`	Anhängen von `s2` an `s1`
`strcmp(s1,s2)`	Lexikographischer Vergleich der Strings `s1` und `s2`
`strcpy(s1,s2)`	Kopiert `s2` auf `s1`
`strlen(s)`	Anzahl der Zeichen in String `s` (= `sizeof(s1)-1`)
`strchr(s,c)`	Sucht Character `c` in String `s`

Grundlage der Programmierung - Ausdrücke

- **Bedingte Ausdrücke**

```
if (Bedingung) {                // (Bedingung)?(TRUE):(FALSE);
// Code wenn TRUE
}
else {
// Code wenn FALSE
}
 Klammern sind optional: Siehe unten
if ( x < 0 ) x = -x; //Bedingung
y = -y; // Immer ausgeführt
```

Grundlage der Programmierung - Schleifen

- **while Schleife**

  while (Bedingung) {
  // Code
  }

- **do-while Schleife**

  do {
  // Code
  } while (Bedingung)
  mindestens 1 Durchlauf

Grundlage der Programmierung - Schleifen

- **for Schleife**

```
for (init-statement; test-expr; increment-expr) {
// Code
}
```

- **For Schleife mit break und continue Ausdrücke**

```
for (i = 0; i < 100; i++ ) {
if ( i == j ) continue;
if ( i > j ) break;
}
```

Grundlage der Programmierung - Arrays

- **Arrays**

  Kollektion von Elementen des gleichen Typs

  Deklaration

  float x[100]; // „Vektor“

  Abrufen der Elemente

  x[0]; // erstes Element

  x[99]; // letzes Element

  Initialisierung

  float x[3] = {1.1, 2.2, 3.3}; oder auch floatY[] = {1.1, 1.2, 1.3};

  Mehrdimensionale Arrays

  float x[4][4]; // Matrix [Zeilen][Spalten] engl. [Rows][Cols]

  int m[2][3] = { {1, 2, 3},{4, 5, 6} };

  Für m[2][3] ist {1,2,3,4,5} gleich {{1,2,3},{4,5,0}}; Es wird empfohlen, bei mehrdimensionale Arrays, mindestens die Spalten Anzahl anzugeben.

Grundlage der Programmierung - Zeiger

- Problem:
  - Variablenname ist fest mit Speicherbereich verbunden.
  - Ziel: Programmstück, das beliebige Speicherbereiche verarbeiten kann, ohne vorher extra eine Kopie zu machen (vorausgesetzt, der Typ stimmt).
- Beispiel
  - Annahme: Polynom ist struct mit 100 Koeffizienten

Problem:

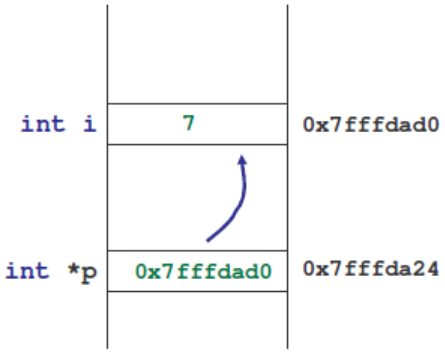
```
Polynom ptmp, p1, p2;
if (bedingung)
   ptmp = p1;                  // kopiert 100 Koeff.!
else
    ptmp = p2;                 // dito
bearbeite ptmp
wieder zurück kopieren;        // kopiert 100 Koeff.!
```

Lösung:

```
Polynom p1, p2;
Polynom-Zeiger ptmp;
if (bedingung)
    ptmp zeigt jetzt auf p1
else
    ptmp zeigt jetzt auf p2
Bearbeite das worauf ptmp zeigt
```

Grundlage der Programmierung - Zeiger

- Bislang griffen wir stets direkt auf Variablen zu, d.h., es war nicht von Interesse, wo die Daten im Speicher abgelegt sind. Ein neuer Variablentyp, der Pointer (Zeiger), speichert Adressen unter Berücksichtigung des dort abgelegten Datentyps.
- **Ein Zeiger**
  - Variable, wie alle anderen auch (z.B int *p)
  - Steht irgendwo im Speicher an bestimmter Adresse
  - Hat einen Wert.
  - Bedeutung des Wertes = Adresse einer anderen Variable



- Eigenschaften
  - Auf Wert einer anderen Variable zugreifen, ohne deren Name zu verwenden (oder kennen)! Ansonsten fast alle Fähigkeiten der normalen Variablen

Grundlage der Programmierung - Zeiger

- Sei der **Zeiger** auf ein **Objekt** vom **Typ int** mit **p** bezeichnet, so ist

  int *p;

  dessen Deklaration, oder allgemein wird durch

  [speicherklasse] <typ> *<bezeichner>;

  ein Zeiger auf den Datentyp <typ> definiert.

  So können die folgenden Zeigervariablen definiert werden

```
Struct Student{
…
};
// Pointer deklaration
char *cp;       // Zeiger auf char
int x, *px;     // Zeiger auf int
float *fp[20]; // Array mit 20 Zeiger auf float
float *(fap[10]); // Zeiger auf Array mit 10 float
Student *ps;  // Zeiger auf structure Student
char **ppc;    // Zeiger auf Zeiger von char
```

Grundlage der Programmierung - Zeigeroperatoren

- Der unäre **Referenzoperator** (Adressoperator)

  &<variable> z.B wie unten: &i

  bestimmt die Adresse der Variablen im Operanden.

- Der unäre **Dereferenzoperator** (Zugriffsoperator)

  ***<pointer> z.B wie unten *pint**

  erlaubt den (indirekten) Zugriff auf die Daten auf welche der Zeiger zeigt. Die Daten können wie eine Variable manipuliert werden.

```
int main()
{
int i, j, *pint;
i = 10;          // i = 10
pint = &i;       // Zeiger Initialisierung
j = *pint;       // zugriff auf int
*pint = 0;       // i = 0
*pint += 2;      // i += 2
return 0;
}
```

Grundlage der Programmierung - Zeiger auf Struktur

```
struct Student
{
...
};
Student peter, *pg;
//      init peter
...
pg = &peter;                    // Zeiger auf peter
cout << (*pg).vorname;          // Konventionale Zugriff
cout << pg->vorname;            //bessere Zugriff
...
```

- Die Zugriffe (*pg).vorname und pg->vorname sind völlig äquivalent. Allerdings verbessert letzterer deutlich die Lesbarkeit eines Programmes insbesondere, wenn der Zeiger ein dynamisches Feld des Typs Student darstellt.
- Dies zeigt sich insbesondere beim Zugriff auf Feldelemente von vorname (d.h., einzelne Zeichen). Der Zugriff auf das **0. Zeichen** erfolgt mittles pg->vorname[0] oder *pg->vorname oder (*pg).vorname[0] oder *(*pg).vorname und der Zugriff auf das 3. Zeichen mittels pg->vorname[3] oder *(pg->vorname+3) oder (*pg).vorname[3] oder *((*pg).vorname+3)

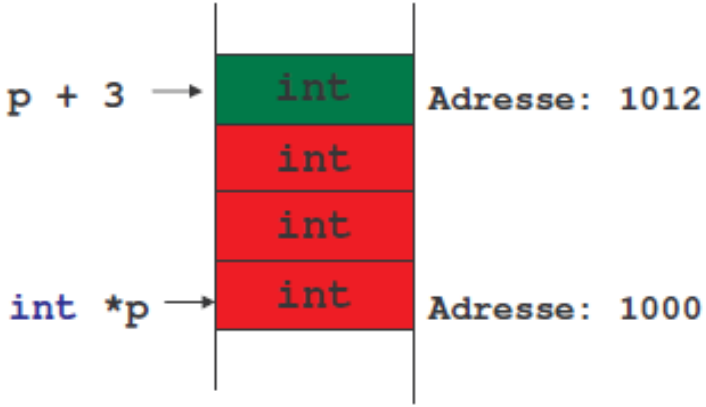
Grundlage der Programmierung - Zeiger Arithmetik

- Hier speilen die Datentypen eine große Rolle.

Jeder **int** Wert hat **4 Bytes** in der Speicher. C++ Ausdruck sizeof(int) = 4

Zeiger + Typ = wert*sizeof(Typ)

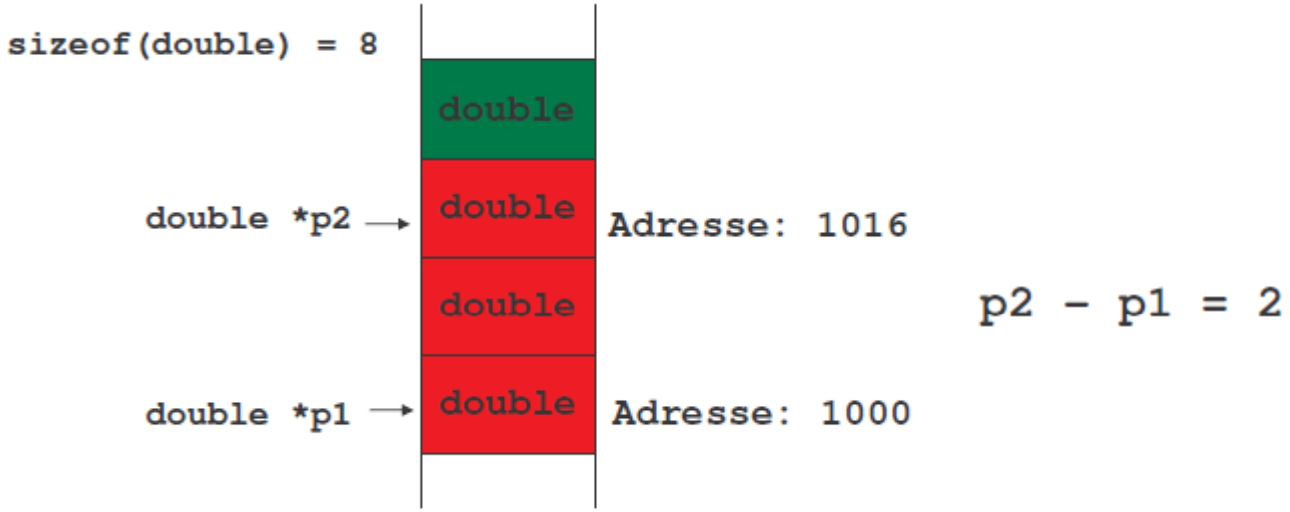
Also: Zeiger + int k=> Zeiger + **k*sizeof(int)**



Grundlage der Programmierung - Zeiger Arithmetik

- **Zeiger Subtraktion**

Zeiger – Zeiger bedeutet (Adresse –Adresse)/ sizeof(Typ)



Grundlage der Programmierung - Null Zeiger

- **Problem:**

wie unterscheidet man gültige Zeiger von Zeigern, die auf nichts zeigen soll?

- Adresse 0 bzw. Wert NULL ist genau dafür reserviert.
- Was passiert, wenn man Null-Pointer dereferenziert?
  - Core Dump (relativ einfacher Bug)
  - Passiert oft auch bei uninitialisierten Zeigern.

```
int * c = NULL;
std::cout << c << std::endl;          // OK
std::cout << *c << std::endl;         // core dump
```

Grundlage der Programmierung - Referenzen

- Eine Referenz ist ein Alias (Pseudoname) für eine Variable und kann genauso wie diese benutzt werden. Referenzen stellen (im Gegensatz zu Zeigern) kein eigenes Objekt dar, d.h., für sie wird kein zusätzlicher Speicher angefordert.

```
main()
{
int i;          // i
int &ri = i;    // deklaration von Referenz auf  i
int *pi;
pi = &i;        // deklaration von Zeiger auf   i;
i = 7;
cout << i << ri << *pi;
ri++;
cout << i << ri << *pi;
(*pi)++;
cout << i << ri << *pi;
}
```

Paradigmen

- Der Schwerpunkt liegt hier auf der Vermittlung **objektorientierter Programmiermethoden,** in einfacher und anschaulicher Form.
- **Wichtig!** Ob die Programmiersprache nun C++, C#, Java, Visual Basic oder einen anderen Namen trägt, ist sekundär; Wichtig ist das Verständnis der dahinter stehenden Grundkonzepte. Wir werden unsere Beispiele in C++ verführen.
- Objektorientierte Programmierung ist ein Paradigma.
- **Paradigma**: Ein Beispiel, das als Muster oder Modell dient.
- Vier vorwiegend bekannte Haupttypen:
  - Prozedurale/Imperative – Strukturierte Programmierung
  - Logische
  - Funktionale
  - **Objektorientierte**

Paradigmen

- Bevor wir mit objektorientierten Programmier-Paradigma anfangen, wollen wir zuerst eine Einführung in die Programmier-Paradigmen im Allgemeinen geben.
- Viele von Ihnen haben bisher nur prozedurale programmiert.
- **Imperative /prozedurale Paradigma:** Die Funktionen stehen im Vordergrund. Mit bedingten Anweisungen und Sprung-Anweisungen können Programmteile übersprungen oder wiederholt werden.
- Bei prozeduralen Programmiersprachen werden ***zu lösende Probleme*** in Teilprobleme aufgeteilt - auch ***Funktionen*** (C/C++) bzw. ***Prozeduren*** (Modula, PASCAL) genannt.

Paradigmen

- **Funktionale- Paradigmen:** Menge von Funktionsdefinitionen und einem Ausdruck. Typischer Vertreter ist die Sprache LISP.
- **Logische-Paradigmen:** Hier werden nur Fakten und Regeln angegeben. Problemlösung wird nicht genauer spezifiziert, sondern vom Interpreter-Programm erstellt. Man kann Logische Paradigmen in Prolog und SQL sehen.
- **Unterschiede:**
  - Funktionale und logische Stille trennen sehr klar die **WELCHE Aspekte** eines Programms (Programmierer Verantwortung) und die **WIE Aspekte** (Durchführungsbeschlüsse).
  - Imperative/prozedurale und objektorientierte Programme enthalten im Gegensatz sowohl die **Spezifikation** und die **Details der Implementierung,** sie sind untrennbar miteinander verbunden.

Paradigmen-objektorientiert

- Hier stehen die Daten (Eigenschaften) und nicht die Funktionen oder Prozeduren des Programms im Vordergrund .
- Die Daten werden in Objekten gekapselt (Information hiding), die **auch** über Funktionalitäten verfügen, um die Daten zu ändern.
- In diesem Zusammenhang spricht man jedoch nicht von **Funktionen**, sondern von **Methoden**.
- Programme werden aus verschiedenen Objekten aufgebaut.
- Beispiel-Sprachen:

  C Sprache wurde im Hinblick auf objektorientierte Programmierung zu C++ weiterentwickelt. C++ ist hybrid aus imperativem/Prozeduralem C und objektorientierten Erweiterungen aufgebaut.
- Heute ist Java neben C++ „state of the Art“.

OO Grundlagen

- Der Trick, den die OOP anwendet, besteht darin, zusammengehörende Daten und Funktionen, die auf diesen Daten operieren, in Objekte zu kapseln. D.h. Verallgemeinerung von Eigenschaften und Funktionalität.
- OO Grundkonzepte:
  - **Modularisierung** - große Software-Projekte lassen sich in kleinere Stücke geteilt.
  - **Wiederverwendbarkeit** - Programme können von schongeschrieben Softwarekomponenten zusammengesetzt werden.
  - **Erweiterbarkeit** - Neue Software-Komponenten können geschrieben oder aus bestehenden weiterentwickelt werden.
  - Objektdaten **Abstraktion** bezeichnet den Prozess, die für eine Darstellung wesentlichen Attribute oder Merkmale über etwas herauszufinden und die unwesentlichen wegzulassen.
  - **Datenkapselung:** Daten (Attribute) sind nicht sichtbar aber Methoden zur Manipulation der Attribute sind sichtbar.

Implementierung mit C++

- C++ ist eine Weiterentwicklung der strukturierten prozeduralen Programmierung in C
- C++ unterstützt ein Programmierparadigma, das auf der Objektorientierung basiert
- In C++ werden objektorientierten Konzepte (Klassen, Objekte, Vererbung, Polymorphie, etc.) in Programmcode direkte Umgesetzt

C und C++ - was ist neu in C++?

- Templates :
  - Zur „Parametrisierung“ von Klassen oder Funktionen
- Exceptions
  - Zur standardisierten Fehlerbehandlung, ähnlich wie in Java
- Namespaces
  - Ähnliches Konzept wie Packages in Java; vor allem um Namenskollisionen zu vermeiden (z.B. zwei mal denselben Namen für eine Klasse), relativ neu in C++

C und C++ - was ist neu in C++?

- Syntaktische Details
  - Datentyp bool, Kommentare mit //, Datentypkonversion mit Funktionsschreibweise type(...), Deklaration von Variablen nicht nur am Anfang eines Blocks, Default-Werte für Argumente in Funktionen/Methoden
- C++ Standard Library
  - Komplette C Standard Library, plus neue Funktionalität
  - iostream Library: Als Eratz (oder besser als zweite Möglichkeit) der Input/Outpt Funktionen (stdio.h) der C Standard Library
  - Standard Template Library (STL): Klassen für vectors, queues, lists etc.

Objektorientierte Programmierung

- In Verbindung mit dem Prinzip der Abstraktion führt das Konzept der Kapselung zu Softwarebausteinen, die ingenieurmäßig zu großen Softwaresystemen zusammen gesetzt werden können.
- Kapselung bedeutet, dass die Daten (Attribute) mit Operationen assoziiert sind, die - und nur die - auf sie zugreifen können. Die Klassen sind die Kapseln.
- OOP erfüllt mit der Unterstützung von **Modularisierung**, **Wiederverwendbarkeit**, **Erweiterbarkeit**, **Abstraktion** und **Kapselung** von **Objekte** die zentralen Anforderung, um die immer komplexere Anwendungslandschaft beherrschbar zu machen

OOP-Vorgehensweise

- Identifikation von Objekten und Implementierung von Objekt-Typen in Klassen.
- Zuweisung von Aufgaben zu diesen Objekten (Methoden).
- Erzeugte Objekte von Klassen in **Main**, kommunizieren mit anderen Objekten gleichen Typs durch Senden von Nachrichten über Methoden.
- Die Nachrichten werden ebenfalls von den Methoden des Objekt-Typs empfangen und verarbeitet.
- Konkret:
  - Ohne objektorientierte Konzepte werden zum Beispiel, Matrizenwerte durch eine Sammlung von einzelnen Variablen(Daten) und Funktionen(Operationen) repräsentiert.
  - Die Beziehung zwischen diesen Daten und Funktionen wird vor allem nur in Ihrem Kopf hergestellt.

OOP- Grob erklärt!!!

- Was bedeutet nun objektorientiert?
  - Bei der objektorientierten Entwicklung werden die in der realen Welt vorkommenden Gegenstände und Begriffe als Objekte betrachtet.
- Objekt
  - „Gegenstand“, der im allgemeinen aus dem Vokabular des Problem- oder Lösungsbereichs stammt.
- Klasse
  - Zusammenfassung mehrerer Objekte zu einer Klasse. Eine Klasse definiert gleichartige Objekte, beschreibt Gemeinsamkeiten im Verhalten und in der Struktur und liefert den „Bauplan“ für ein Objekt.
- Objekt vs. Klasse
  - Ein Objekt ist eine eindeutige, konkrete Einheit, die in Zeit und Raum existiert, während eine Klasse die Abstraktion des “Wesentlichen” von Objekten an sich ist.

Grundbegriffe- Objekte

- Ein "Objekt" ist alles, was diesem Konzept entspricht,

  oder

- ein Objekt repräsentiert ein Individuum, identifizierbaren Artikel, Einheit, oder Rechtsträger, entweder real oder abstrakt, mit einer klar definierten Rolle bei der Problem-Domäne.

  Beispiele:

  - Materielle Dinge - wie ein Auto, Drucker, ...
  - Rollen - als Mitarbeiter, Chef, ...
  - Vorfälle - wie Flug-, Überlauf-, ...
  - Interaktionen - als Vertrag, den Verkauf, ...
  - Technische Daten - wie Farbe, Form, ...

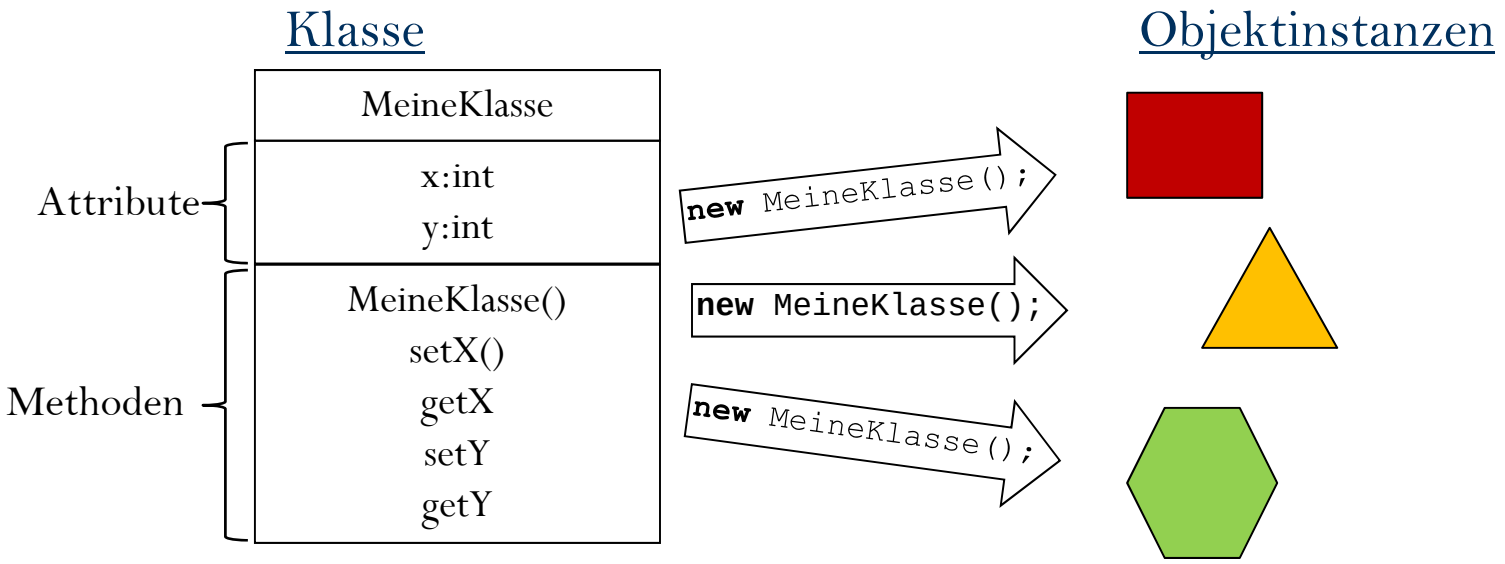
Grundbegriffe- Objekte

- Die zwei Teile eines Objekts
  - Objekt = Daten + Methoden

- In anderen Worten: Ein Objekt hat die Verantwortung, zu wissen und die Verantwortung zu tun. Im Gegensatz dazu obliegen in der Sprache C diese Aufgaben dem Programmierer.

Grundbegriffe-Klassen

Objekte basieren auf Klassen, die einen Bauplan für ein Objekt und dessen Attribute(Daten) und Methoden festlegen.



Klassen fassen gleichartige Objekte zusammen.

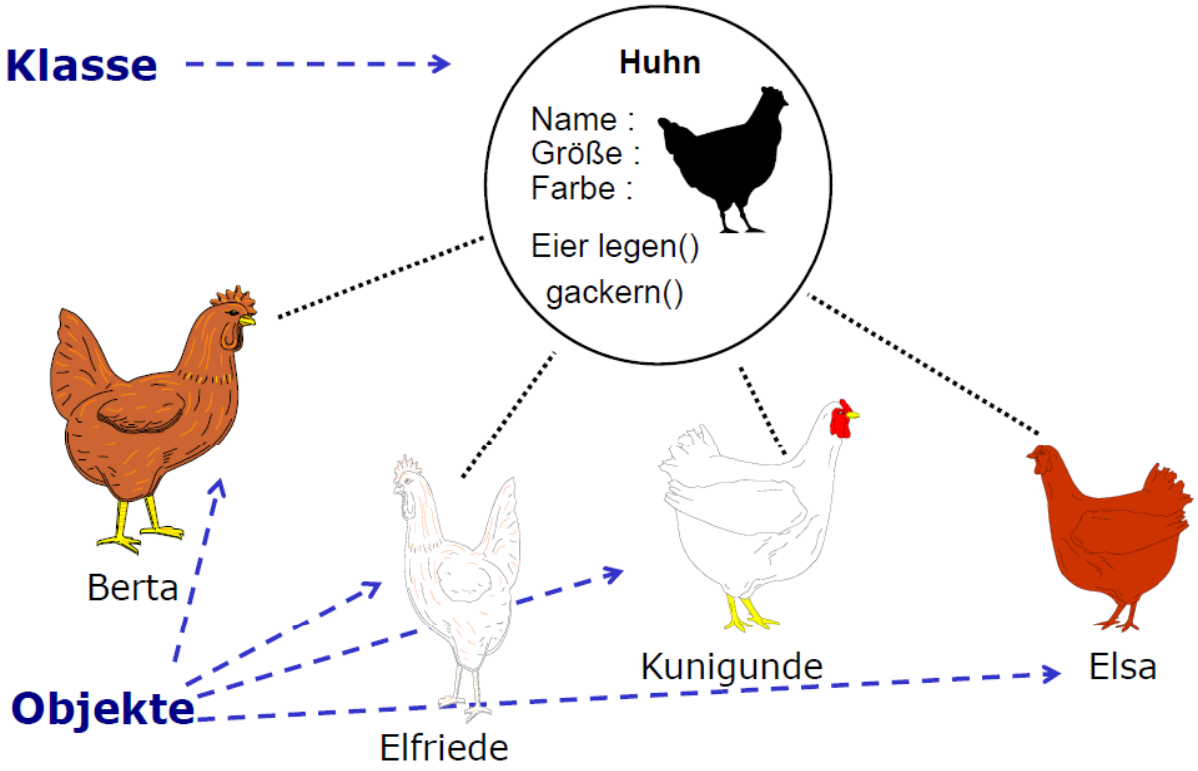
Grundbegriffe – Objekte/Klassen

Zusammenfassung:

- Objekte = Daten + Methoden
- Daten (Attribute) = Eigenschaften
- Methode = Operationen rund um das Objekt
- Klassen (Objektkapsel) = Program-Bauplan (Implementierte Objekt in einer Programmiersprache)

Beispiel Objekte: Siehe Beispiele …

Grundbegriffe – Objekte/Klassen



C++ Klassengerüst

Gerüst (Grob)

```
class Demo
{
private:
  // private Eigenschaften und Methoden
protected:
  // geschützte Eigenschaften und Methoden
public:
  // öffentliche Eigenschaften und Methoden
};
```

Gerüst mit Inhalt

```
class Demo{
private:
      Datentyp name;                    //Instanzvariable
      Static Datentyp name;//Klassenvariable
public:
      Demo(Datentyp parameter); //Konstruktor
      Datentyp Funktion1();             //Memberfunktion
      Datentyp Funktion2(Datentyp parameter);
      Datentyp Funktion3() const;       // const-Memberfunktion
      static Datentyp Funktion4() //static-Memberfunktion
};
```

Klassen und Strukturen in C++

Die Basis der OOP

- C++ erlaubt unter anderen die Deklaration von Klassen und Strukturen zur Strukturierung eines OOP-Programms.
- Klassen und Strukturen unterscheiden sich in C++ prinzipiell kaum. In beiden können Sie Eigenschaften und Methoden unterbringen.
- Klassen werden mit dem Schlüsselwort class deklariert, Strukturen mit struct.
- Alle Datenfelder in einer Struktur sind öffentlich. Klassen ermöglichen dagegen, dass Datenfelder privat oder geschützt (protected) deklariert werden. Strukturen ermöglichen somit nicht das wichtige Konzept der **Kapselung.**

```
class person{
        string Name;  // private
        string Nachname   //private
      public:
        void laufen(){       //public
        }
};
```

```
struct  person{
        string Name;  // public
        string Nachname   //public
        void laufen(){       //public
        }
};
```

Grundbegriffe – Objekte/Klassen

- Bereicherunf!!!

```
class X {
// Mitglieder von  X  Klasse kommen hier
};
struct Y {
// Mitglieder von  Y struct kommen hier
};
union Z {
// Mitglieder von  Z union kommen hier
};
//Main Methode
int main() {
X xobj; // deklaration eine class Objekt der Klasse Typ X
Y yobj; // deklaration eine struct Objekt der Klasse Typ Y
Z zobj; // deklaration eine union Objekt der Klasse Typ Z
}
```

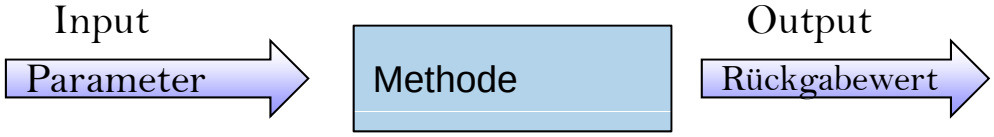
Klassen und Strukturen in C++

- Strukturen können nur wie einfache Klassen behandelt werden, damit die Umsetzung von C-Quellcode nach C++ vereinfacht wird. Da die Strukturen die wichtige Kapselung nicht unterstützen, sollten Sie stattdessen immer Klassen einsetzen.

Grundbegriffe - Methoden

- Eine Methode ist eine Funktion/Operation, die innerhalb und in bestimmten Fällen auch außerhalb eines Objekt-Typs (Klasse) agiert. Sie berechnet einen Ergebnis-Wert aus Argument-Werten und können dabei auf die Bestandteile des Objekts zugreifen.
- Die **Deklaration** einer Methode ist Bestandteil der Deklaration eine Klasse:

```
class Klassenname {
...
Zugriffsrecht:                                  //meisten public für Methode
    rückgabetyp  Methodenname (parameter); //Methoden Deklaration
 ...
};
```

- Rückgabetyp und Parameter sind Datentypen.
- 

Grundbegriffe - Methoden

- Da Methoden zu den Objekt-Variablen einer bestimmte Klasse gehören, muss bei ihrer Definition der Klassenname angegeben werden:
- **Definition:**

```
rückgabetyp Klassenname::Methodenname () {
...
anweisungen;
return wert;
} // Methoden Definition <- Typisch C++.
```

- Der Aufruf einer Methode ist ihre Anwendung auf eine Objekt-Variable.
- Die Methode einer Klasse kann auf jede Objekt-Variable diese Klasse angewendet werden.
- **Methodenaufruf:**

  Methodenname(argumente);

  Var = Methodenname(argumente);

Grundbegriffe – spezielle Methoden

- **Main-Methode (Projekt-Haupteingang):** Jedes OOP-Programm muss eine Methode mit der Bezeichnung Main enthalten.
- Ohne Main, läuft ein Program nicht!
- **Die 2 gültigen C + + und bekannten main-Signaturen:**

  int main() <-vorläufig werden wir nur diese benutzen, bis Sie Zeiger (Pointers) verstehen.

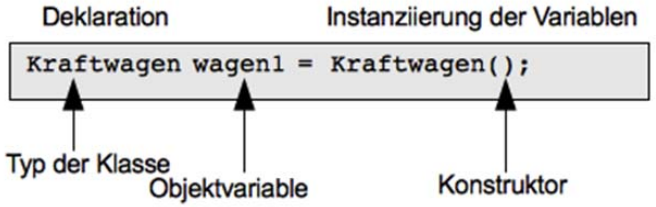
  int main(int argc, char* argv[])
- **Obwohl viele Compiler auch diese Signatur akzeptieren**

  int main(int argc, char** argv)
- void main() wird nicht in C++ unterstützt!
- Ein Programm soll mit einem 0-Rückgabewert von „main“ beenden. Sollte es nicht explizit angegeben sein, erzeugt der Compiler entsprechenden Code automatisch.

  Return 0;

Grundbegriffe – spezielle Methoden

- **Konstruktoren:** Sie steuern die Erstellung/Erzeugung eines Objektes über die Main Methode. Es sind spezielle Methoden, die bei der Instanziierung von Objekten (Erzeugen eines Objektes der Klasse – s.u.) aufgerufen werden.
- Ein Konstruktor hat immer den gleichen Namen wie die Klasse.



- Konstruktoren ohne Parameter werden als Default- oder Standardkonstruktoren bezeichnet.
- Wenn in einer Klasse explizit kein Konstruktor definiert wird, definiert der Compiler implizit eine Defaultkonstruktor.
- Beispieldefinition:

```
Kraftwagen(){
//Definition hier
}
```

Grundbegriffe – spezielle Methoden

- **Destruktoren:** Nachdem das Objektes über Konstruktor-Aufruf erzeugt wurde, kann beliebig mit dem Objekt gearbeitet werden. Wird das Objekt nicht mehr benötigt, kann es über den Destruktor „zerstört“ werden. Dadurch wird der belegte Speicherplatz wieder frei.
- Es gibt auch einen Default/Standard Destruktor. Wird immer implizit aufgerufen.
- **Rufen Sie nie ein Standard-Destruktor explizit auf!**
- Destruktoren kann man auch explizit definieren.
- Hat auch immer den gleichen Namen wie der Klassenname.
- Beispieldefinition:

```
~Klassenname(){
//Definition hier
}
```

Deklaration und Definition

- Definition und Deklaration sind zwei verschiedene Dinge in C++ oder in der Programmierung überhaupt.
  - Eine Deklaration informiert den Compiler über die Eigenschaften etwa von einem Typen oder einer Funktion, definiert aber keinen Code, der im ausführbaren Programm verwendet wird. Also namens Gebung von Variablen und Methoden.
  - Wenn wir ein struct/Method/Klasse/Variable deklarieren, deklarieren wir im Grunde nur ein Muster.

```
void hallo(); // Deklaration von "hallo" (ohne Definition)
```

  - Eine Definition andererseits, definiert etwas, das im ausführbaren Programm tatsächlich existiert, etwa eine Variable oder Code-Zeilen.

```
void hallo(){ std::cout << "Hallo!\n"; }   // Definition von "hallo"
```

```
int main(){
hallo ();
return 0;
 } // Verwendung von "hallo„ in main
```

Grundbegriffe - Datentypen

- **Variablen** werden verwendet, um **Daten** zu speichern. Eine Variable hat einen **Namen** und den **Datentyp**.
- Global gibt zwei Arten von Datentypen

  Primitive (vorgegeben) und Nicht-Primitive (Benutzerdefiniert).
- **Benutzerdefinierte Datentypen:** Wenn Sie eine Klasse definieren, definieren Sie im Prinzip einen eigenen Datentyp. Dieser Datentyp kann jedoch nicht nur über Eigenschaften, sondern auch über eigene Methoden verfügen.
- Beispiel primitive Datentypen in C++ (Wichtig: die werden klein geschrieben):
  - bool //Wahrheitswerte: true und false
  - char // Einzelnes Zeichen: 'a', '!', '\n,
  - int //Ganzzahlen im Bereich von -32768 bis 32767
  - long //Ganzzahlen im Bereiche von -2147483648 bis 2147483627
  - float //Beliebige Zahlen im Bereich von -3.40e+38 bis 3.40e+38
  - double // Beliebige Zahlen im Bereich von -1.79e+308 bis 1.79e+308

Grundbegriffe - Datentypen

- **Varieblendeklaration: primitive Datentypen**
  - int meineVar; //einfache Deklaration
  - double Var1, Var2; //mehrere Variablen eines Typs
  - char zeichen = 'a' //Deklaration mit Initializierung

  Beispiele von Oben: Benutzerdefinierte Datentypen
  - X xobj; // deklaration eine class Objekt der Klasse Typ X
  - Y yobj; // deklaration eine struct Objekt der Klasse Typ Y
  - Z zobj; // deklaration eine union Objekt der Klasse Typ Z

Grundbegriffe - Zugriffsrechte

- **Zugriffsrechte (Sichtbarkeit**): Programmiersprachen erlauben eine abgestufte Sichtbarkeit (oder Zugriffsrechte) für Daten (Attribute) und Methoden und Klassen.
- Die wichtigsten Zugang Typen sind **Public** , **Private** , **Friend** und **Protected**.
- **Private:**
  - Mit private stellen Sie die Sichtbarkeit eines Elements so ein, dass dieses nur innerhalb der Klasse gilt. Alle Methoden der Klasse können auf private Elemente zugreifen. Von außen können solche Elemente allerdings nicht verwendet werden. Wenn Sie damit eine Eigenschaft deklarieren, handelt es sich im Prinzip dabei um einfache Variablen, die von allen Methoden der Klasse verwendet, aber nicht von außen gesetzt oder gelesen werden können.
- **Protected:**
  - Der Modifizierer protected kennzeichnet Klassenelemente, die zunächst wie private Eigenschaften auftreten, aber bei der Vererbung in abgeleiteten Klassen verwendet werden können (was bei privaten Elementen eben nicht möglich ist).

Grundbegriffe - Zugriffsrechte

- **Public:**
  - Elemente, die den Modifizierer **public** besitzen, können von außen (über eine Referenz auf ein Objekt dieser Klasse oder ? bei statischen Elementen ? über den Klassennamen) verwendet werden.
- **Friend:**
  - Dieses Zugriffsrecht wird bei der Definition der Klasse verliehen, auf deren Bestandteile zugegriffen werden soll. Die Klasse **selbst** legt fest, wen sie zum Freund haben will. Eine nachträgliche "Anbiederung" ist nicht möglich.

Die Strukturierung einer Anwendung

- Größere Programme erfordern eine Strukturierung des Quellcodes. Die Basis der Strukturierung ist in C++ eine Klasse.
- Über Klassen erreichen Sie, dass Sie Programmcode wiederverwenden können und dass Sie Ihr Programm so strukturieren, dass die Fehlersuche und die Wartung erheblich erleichtert werden.
- Wenn Sie eine Klasse entwickeln, können Sie entscheiden, ob Sie eine **echte** Klasse (mit **normalen** Eigenschaften und Methoden) programmieren, aus der Sie später Instanzen erzeugen, oder ob Sie eine Klasse mit statischen Methoden und Eigenschaften erzeugen wollen.
- Statische Methoden und Eigenschaften (Klassenmethoden, Klasseneigenschaften) können Sie auch ohne eine Instanz der Klasse aufrufen.
- **Echte** Klassen arbeiten echt objektorientiert, Klassen mit statischen Methoden und Eigenschaften simulieren (u. a.) die Module der strukturierten Programmierung.

Einfache Klassen und deren Anwendung

Grundlagen zur Programmierung von Klassen

- In einer C++-Datei (mit der Endung *.cpp* oder *.cc*) können Sie eine oder mehrere Klassen implementieren.
- Sie können Klassen komplett in der *cpp*-Datei unterbringen und auf eine Headerdatei verzichten oder
- In *h*-Datei und *cpp*-Datei trennen. Die *h*-Datei (Headerdatei) enthält die Deklaration der Klasse, die *cpp*-Datei enthält die Implementierung der Methoden der Klasse.
- In einer *cpp*-Datei können Sie eine oder mehrere Klassen unterbringen.
- In der Praxis werden Klassen oft in separaten Dateien deklariert, um diese einfach in anderen Programmen wiederverwenden zu können.
- Diese Trennung macht auch dann Sinn, wenn Sie Ihre Klassen in Bibliotheken (mit der Endung *.lib*) kompilieren und diese Bibliotheken für die eigene Verwendung in einem separaten Ordner speichern.

Die Strukturierung einer Anwendung

Variante 1 - Trennung

```
#include <iostream>
using namespace std;
Class Demo{
//Deklaration
Public:
      Demo(Datentyp parameter); //Konstruktor
      Datentyp Funktion1();                  //Memberfunktion
      Datentyp Funktion2(Datentyp parameter);
      Datentyp Funktion3() const; // const-Memberfunktion
      static Datentyp Funktion4() //static-Memberfunktion};
Private:
    int Radius;
//Definition
Demo::Demo(Datentyp parameter) // Konstruktor Definition
{
Anweisung
}
Datentyp Demo::Funktion1() //Methoden Definition
{
Anweisung
}
...usw
```

Variante 2- alles zusammen

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

Class Klassenname{

Private:
      int Radius;

//Gleichzeitig Deklaration und Definition
Public:
      Demo(Datentyp parameter){
         Anweisung
        }
      Datentyp Funktion1(){
         Anweisung
        }

      Datentyp Funktion2(Datentyp parameter){
         Anweisung
        }
        ...usw
};
```

Einfache Klassen und deren Anwendung

Objekte statisch erzeugen in der Main Methode

- Statisch erzeugen Sie ein Objekt, indem Sie eine Objektvariable wie eine einfache Variable deklarieren:

```
Demo  d;      // statische Erzeugung
```

- Statisch bedeutet hier, dass C++ das Objekt auf dem Stack speichert. Der Stack ist ein Speicherbereich, den alle Programme besitzen und auf dem u. A. alle lokalen Variablen einer Funktion oder Methode gespeichert werden.
- Da der Stack nach der Ausführung der Funktion/Methode wieder automatisch bereinigt wird, werden statische Objekte also automatisch zerstört, wenn die Funktion bzw. Methode beendet ist.
- Ein Problem bei statischen Objekten ist, das Objekt könnte zu groß sein, und man muss bei der Entwicklung immer beachten wie viele Objekt erzeugt werden müssen.

Einfache Klassen und deren Anwendung

Objekte dynamisch erzeugen in der Main Methode

- Für die dynamische Erzeugung von Objekten deklarieren Sie die Objektvariable als Zeiger:

```
Demo * d;      // dynamische Erzeugung
```

- Dann erzeugen Sie das Objekt über den new-Operator:

```
Demo  = new Demo
```

- Dynamisch erzeugte Objekte werden auf dem Heap angelegt. Der Heap ist ein Speicherbereich, der für globale Daten reserviert ist. Auf dem Heap werden außerdem alle Funktionen und Klassen einer Anwendung gespeichert. Die Größe des Heap ist nur durch den im System verfügbaren Speicher begrenzt.
- So kann Ihre Anwendung während des Programmablaufs (eben dynamisch) nahezu beliebig viele Objekte erzeugen.

Einfache Klassen und deren Anwendung

Objekte dynamisch erzeugen in der Main Methode

- Objekte, die dynamisch erzeugt wurden, können über eine Dereferenzierung des Zeigers angesprochen werden:

```
(*d). Radius = 20
```

- Durchgesetzt hat sich aber die Kurzschreibweise, die mit dem Operator -> arbeitet:

```
d->Radius = 20
```

- Dynamisch erzeugt Objekte müssen Sie wie alle dynamisch reservierten Speicherbereiche über die delete-Anweisung aus dem Speicher entfernen.

```
delete  d;
```

- Vergessen Sie dies, bleibt das Objekt im Speicher, so lange Ihre Anwendung läuft, und verbraucht unnötig Ressourcen.

Einfache Klassen und deren Anwendung

Die Bedeutung des Stack

- Der Stack ist ein spezieller Speicherbereich, den jedes Programm besitzt und der wie ein Stapel arbeitet: Daten werden der Reihe nach auf dem Stapel abgelegt und können auch wieder, der Reihe nach, vom Stapel entfernt werden. (Last in first out-LIFO)
- **Der Stack wird vom Compiler automatisch über Funktionen angesprochen, die Daten auf dem Stapel ablegen, Daten vom Stapel auslesen und vom Stapel löschen.**

Einfache Klassen und deren Anwendung

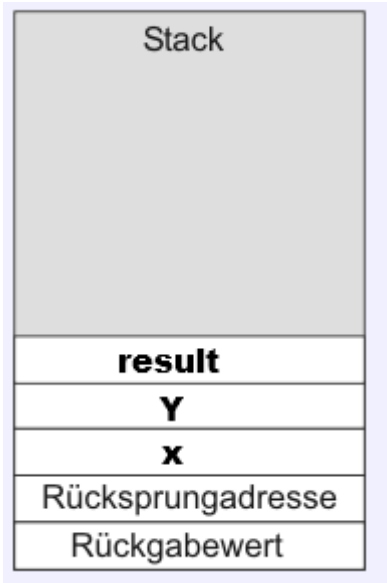
Die Bedeutung des Stack : Beispiel

```
int Add(int x, int y)
{
  int result;
  result = x + y;
  return result;
}

void main(void)
{
  int i = Add(1, 2);
  cout << I;
}
```

Beim Aufruf der Funktion **Add**, legt der Compiler die folgenden Daten auf dem Stack an:

1. einen Speicherbereich für den Rückgabewert der Funktion,
2. die Adresse, zu der das Programm nach der Ausführung zurückspringen muss,
3. die Argumente der Funktion,
4. und die lokalen Variablen der Funktion.



Einfache Klassen und deren Anwendung

Wie werden Objekte gespeichert?

- Um zu verstehen wie z.B den Polymorphismus und virtuellen Methoden funktionieren, müssen Sie wissen wie, Objekte prinzipiell gespeichert werden.
- Eine Instanz einer Klasse besteht prinzipiell nur aus den Datenfeldern der Klasse (außer, wenn die Klasse Inline-Methoden beinhaltet, die innerhalb der Instanz gespeichert werden).
- **Beispiel**

```
class Punkt
{
public:
  int x;
  int y;
  void set(int x, int y)
  {
    this->x = x;
    this->y = y;
  }
};
```

```
class Kreis
{
public:
  int Radius;
  int Farbe;
  double Umfang
  {
    return Radius * 2 * 3.1415927;
  }
};
```

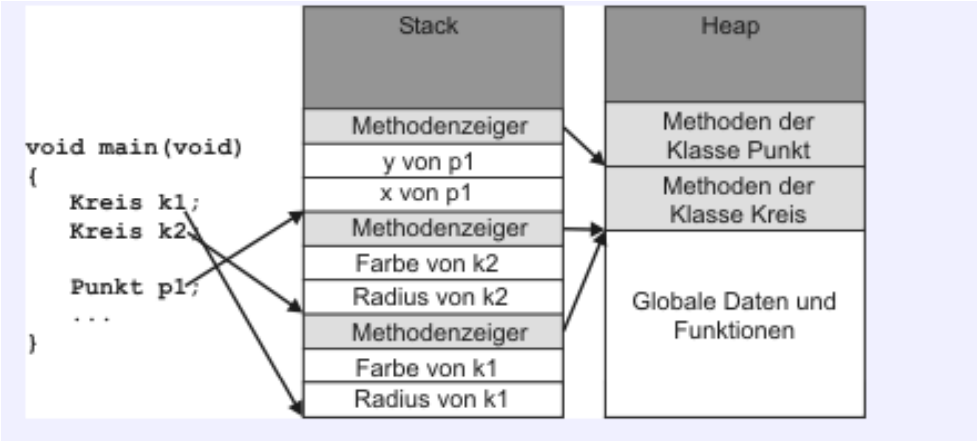
Einfache Klassen und deren Anwendung

Wie werden Objekte gespeichert?

```
void main(void)
{
  Kreis k1;
  Kreis k2;
  Punkt p1;

  k1.Radius = 100;
  k1.Farbe = 255;
  k2.Radius = 200;
  k2.Farbe = 1024;

  p1.Set(10, 11);
}
```



k1, k2 und p1 sind also Variablen, die auf die Adresse zeigen, an der die Eigenschaften des Objekts gespeichert sind. Dieses Vorgehen reduziert den Speicherbedarf von Objekten enorm.

Einfache Klassen und deren Anwendung

Der this-Zeiger

- Damit eine Methode, die ja normalerweise (wenn es keine Inline-Methode ist), die Eigenschaften der aufrufenden Instanz bearbeiten kann, übergibt der Compiler der Methode ein zusätzliches, unsichtbares Argument, den this->Zeiger.

```
double Umfang(Kreis *this)
{
  return this->Radius * 2 * 3.1415927;
}
```

- Dieser Zeiger zeigt auf die Objektinstanz, von der aus der Aufruf erfolgt ist.
- Wenn Sie in einer Methode auf Eigenschaften oder Datenfelder des Objekts zugreifen oder andere Methoden aufrufen, stellt der Compiler diesen Zeiger implizit vor den Namen des verwendeten Klassenelements, falls dies noch nicht explizit geschehen ist und falls ein entsprechendes Klassenelement gefunden wird. Siehe nächste Folie...

Einfache Klassen und deren Anwendung

- **Konkretes this-> Beispiel**

```
class Kreis
{
public:
   int Radius;
   int Farbe;
   double Umfang
   {
      return Radius * 2 * 3.1415927;
   }
};
```

main:

Kreis k1 // Statisch deklariert
K1 als Objekt hat Radius und Farbe als Eigenschaften und kann Umfang (Methode)aufrufen Implizit sieht im kompilierten Programmcode demnach so aus, falls noch nicht so definiert ist

```
double Umfang(Kreis *this)
{
 return this->Radius * 2 * 3.1415927;
}
```

K1 (ist Aufrufer)

this ist der der zu eine gegebene Zeit eine Funktion Ausführt

C++

- Jede Klasse definiert einen neuen Datentyp
- Entsprechend können Variablen dieses Typs deklariert (instanziiert) werden
  - Objekt Instanzen
- Bei der Erzeugung eines Objekts wird ein Konstruktor verwendet
  - Konstruktoren: public Methode mit gleichem Namen wie Klasse ohne Rückgabewert
- In C++ werden Spezifikation und Implementierung einer Klasse üblicherweise getrennt:
  - Spezifikation enthält Variablen und Methodendeklaration(manchmal in einer Headerdatei- .h Datei)
  - Implementierung findet dann außerhalb statt (meistens die .cpp, .C oder .cc Dateien)
  - Man kann auch innerhalb der Klassenspezifikation implementieren (inline): Alles in einer .cpp, .C oder .cc Datei
- *Siehe Innerhalb- und Außerhalb-Beispiele…*

Friend Klassen

- In einigen Fällen kann es notwendig werden, dass andere Klassen oder Funktionen Zugriff auf die geschützten Member einer Klasse benötigen.
- Damit eine Klasse *Class1* Zugriff auf alle Member einer anderen Klasse *Class2* erhält, wird die Klasse *Class1* als *friend*-Klasse der Klasse *Class2* deklariert. Dazu wird innerhalb der Klassendefinition der Klasse, die ihren 'Schutz' aufgibt, folgende Anweisung eingefügt:

  Siehe Beispiel : friend_to_friend

Friend Methoden

- Außer den bisher behandelten *friend*-Klassen gibt es auch noch *friend*-Methoden. *friend*-Methoden gehören keiner Klasse an und haben ebenfalls vollen Zugriff auf alle Member einer Klasse. Um eine Methode als *friend*-Methode einer Klasse zu deklarieren, wird wieder innerhalb der Klasse die ihren 'Schutz' aufgibt folgende Anweisung eingefügt:

  friend Rückgabetyp Methodenname(...);

- *Man beachte:*

  Die *friend*-Eigenschaft einer Klasse ist nicht vererbbar.

- *Siehe Beispiel: friendfunction*

Üben Sie bitte alle Beipiele mit Visual C++ 2010