4. C++, Smalltalk, Eiffel

- C++
  - Historie
  - Funktionen und Programmdateien
  - Streams
  - Klassen (Konstruktoren, Destruktoren, Friends)
  - Copy-Konstruktor, Überladen von Operatoren
  - Abgeleitete Klassen (Vererbung, Virtuelle Funktionen, Abstrakte Klassen, Mehrfachvererbung)
  - Templates
- Smalltalk
- Eiffel
- Vergleich

C++

- C++ ist gegenwärtig die industriell am weitesten verbreitete objektorientierte Sprache
- Entwicklungsziel: Kompatibilität zu C
  - “The language is not called D, since it is an extension of C and does not attempt to remedy problems by removing features”
  - “The better one knows C, the harder it seems to avoid writing C++ in C style”(Bjarne Stroustrup)
- wesentliche Erweiterungen gegenüber C :
  - stärkere Typprüfung
  - Datenabstraktion (Klassen)
  - Objekt-Orientierung (Vererbung, ...)
  - Operator-Overloading
- Historie
  - Grundlagen: BCPL, C, Simula 67; UNIX und UNIX-Programmierumgebung
  - 1979: C with classes
  - 1983: “Adding classes to C: An Exercise in Language Evolution”. Software - Practice and Experience
  - 1986: Bjarne Stroustrup: “The C++ Programming Language”. Addison Wesley
  - 1991: “The C++ Programming Language 2nd ed.” Addison Wesley
Mehrfachvererbung, parametrisierte Typen, Ausnahmebehandlung ...
  - 1994: “ The design and evolution of C++ “ ; Addison Wesley
  - ANSI/ISO-Normung (Namensbereiche, Datentyp bool, ...)

Ein erstes Programm

```
/* Programm zur Berechnung der Dreiecksfläche */
#include <iostream.h>

int main() {
    double Hoehe, Grundseite;
    cout << "Höhe und Grundseite des Dreiecks eingeben !" << endl;
    cin >> Hoehe >> Grundseite;
    double Flaeche = Hoehe*Grundseite*0.5; // Flächenberechnung
    cout << "Fläche des Dreiecks: " << Flaeche << endl;
}
```

- Hauptprogramm: Funktion `main`
- Einfügen von externen Deklarationen aus Header-Dateien: `#include`
- Ein-/Ausgabe unter Nutzung von Bibliotheksfunktionen (hier: Streams)
- Kommentare: Zeilenkommentare (`//`) und Kommentarblöcke (`/* ... */`)
- Namen: Unterscheidung zwischen Groß- und Kleinbuchstaben, keine Längenbegrenzung, keine Nutzung von Schlüsselwörtern

Funktionen

- Funktionsdeklaration: Festlegung der Signatur (Name, Argumenttypen, Rückgabetyp)
  -> Funktionsprototyp

```
void Swap (double* a, double* b);
int   maxDrei (int, int, int);
```

  - Angabe von Parameternamen optional
  - mehrere Deklarationen derselben Funktion in einem Programm möglich
- Funktionsdefinition: Angabe der Implementierung (Rumpf)
- Übergabemechanismus: Call-by-Value (Ausnahmen: Felder und Referenzen)
- Parameter-Besonderheiten
  - konstante Argumente `void Fehler (const char* Meldung) ...;`
  - Standardargumente `void zeigeInt (int i, int breite = 10) ...`

```
zeigeInt (8, 5);
zeigeInt (6);
```

Funktionen (2)

- Inline-Funktionen
  - Schlüsselwort `inline`
  - es erfolgt kein Funktionsaufruf, sondern Compiler fügt Funktionsrumpf an Aufrufstelle ein
  - v.a. für sehr kurze Funktionen günstig, um Aufrufkosten einzusparen

```
inline int Max (int i, int j)
     { return (i > j) ? i : j; }
```

  - inline-Angabe ist lediglich Hinweis an den Compiler
  - Beschränkungen: inline-Funktionen müssen in nutzendem Programm definiert sein, keine rekursiven Aufrufe, keine Schleifen ...

- Überladen von Funktionsnamen
  - mehrfache Definition von Funktionen möglich, wenn unterschiedliche Parametertypen vorliegen (Anzahl, Typ)
  - Auswahl der "richtigen" Implementierung erfolgt über Argumenttypen beim Aufruf
  - Abweichung im Ergebnistyp allein genügt nicht
  - Beispiel: `void Swap (double*, double*)` und `void Swap (int*, int*)` sind unterschiedliche Funktionen

Gültigkeitsbereiche

- C++ unterscheidet verschiedene Gültigkeitsbereiche (scopes)
  - lokal: sichtbar im Block, in dem die Deklaration erfolgt
  - global: sichtbar in allen Blöcken einer Programmdatei (file)
  - Klasse
  - Namensbereich

- Beispiel

```
int x = 50;
main () {
    x++;
    cout << x << endl;
    double x = 10.01;
    x++;
    cout << x << endl;
    { int x = 1;
      x++;
      cout << x << endl;     }
    x++;
    cout << x << endl;
}
```

Interne vs. externe Bindung

- Namen mit *externer Bindung (external linkage)* in mehreren Programmdateien verwendbar
  - Name bezeichnet in allen Dateien dasselbe Objekt, dieselbe Funktion, dieselbe Klasse etc.
  - erfordert Deklaration in nutzender Datei
  - Binder (Linker) muß zugehörige Definition ermitteln

```
extern int a;
int f ();

void g ()
    { a = f ();
    }
```

Datei 1

```
int a = 1;
int f () { /*...*/ ;
         }
```

Datei 2

- Namen mit *interner Bindung* sind nur in der Programmdatei gültig, in der sie definiert sind

Name kann in anderen Dateien zur Bezeichnung anderer Objekte oder Funktionen verwendet werden

- Globale Namen erhalten interne Bindung bei: `static`, `inline` oder `const` (sofern nicht `extern`)
- keine externe Bindung für lokale Namen (blockbezogener Gültigkeitsbereich) außer bei expliziter Angabe von `extern` (und wenn nicht `static`)

Header-Dateien

- Zusammenfassung aller Namen, die in mehreren Dateien benötigt werden (externe Bindung), in Header-Datei
  - Include-Direktive an Compiler zur Einfügung des Dateiinhalts in nutzender Datei
- Header-Dateien sollten keine Definitionen von Funktionen und Objekten (Variablen) enthalten, sondern vor allem
  - Deklarationen von Daten und Funktionen
  - Deklarationen von Klassen, Templates und Typen
  - Definitionen von Konstanten und Inline-Funktionen
  - Include-Direktive
- Programmierer entscheidet, was in welche Header-Datei kommt (z.B. daß obige Emphehlungen eingehalten werden, Verwendung einer oder mehrerer anwendungsspezifischer Header-Dateien ...)
- Header-Dateien führen oft zur Importierung sehr vieler, nicht benötigter Deklarationen

Streams

- Ein-/Ausgabe erfolgt über Standardbibliotheken, v.a. Stream-Bibliotheken

- Stream: abstrakter Datenstrom zwischen einer Quelle und einer Senke
  - Idee: Unwandlung beliebiger Werte und Objekte in Zeichenfolgen und umgekehrt
  - E/A-Operatoren << und >> geben Richtung des Datenflusses an

- Stream-Klassen `istream` bzw. `ostream` zur Eingabe bzw. Ausgabe (`<iostream.h>`)
  - vordefinierte Objekte dieser Klassen: `cin , cout, cerr`
  - E/A für unterschiedliche Datentypen erfolgt durch Überladen der Operatoren << und >>

- Klassen `ifstream` bzw. `ofstream` zur Ein-/Ausgabe mit Dateien (`<fstream.h>`)
  - Angabe des Dateinamens bei Anlegen eines Stream-Objektes
  - Dateioperationen: `open, close,` ...

Streams (2)

- Beispiel für Datei-E/A

```
#include <iostream.h>
#include <fstream.h>

int main () {
    int Hoehe, Breite, Laenge, i;
    char* Dateiname;
    cout << “Dateiname eingeben “ << endl;
    cin >> Dateiname;
    ifstream Eingabe (Dateiname);

    if (!Eingabe)       // Fehler waehrend Oeffnen
      { cerr << “Ungueltiger Dateiname !” << endl;
          return -1; }

    for (i=0; Eingabe  >> Breite >> Laenge >> Hoehe; ++i)
        cout << “Das Volumen betraegt “ << Hoehe*Breite*Laenge << endl;

    Eingabe.close();
    cout << endl << “Anzahl der Eingaben: “ << i << endl;
}
```

Klassen

- Klassen sind benutzerdefinierte Datentypen
- Klassen bestehen aus Klassenelementen (members):
  - Datenelemente
  - Elementfunktionen (member functions)
  - eingebettete Typen (andere Klassen oder Typdeklarationen)
- nur explizit als `public` bezeichnete Elemente sind von außen ansprechbar
- Beispiel

```
class Datum {      int Tag, Monat, Jahr;
      public:      Datum (int, int, int);
                   void next ();
                   void print (); };
```

```
void Datum::print () {
            cout << Tag << ". " << Monat <<
                  ". " << Jahr << endl; }
```

- Einschränkungen
  - Spezifizierer `const, static, extern` für Klasse nicht möglich
  - direkte Initialisierung von Datenelementen nicht möglich

Klassen (2): Konstruktoren

- dienen zur Initialisierung von Objekten einer Klasse
- Konstruktor hat keinen Rückgabetyp und ist i.a. als `public` deklariert
- Konstruktor kann beliebig oft für unterschiedliche Argumenttypen überladen werden

```
class Datum ...
  public     Datum (int, int, int); // Tag, Monat, Jahr
             Datum (int, int);     // Tag, Monat, aktuelles Jahr
             Datum (int);          // Tag, aktueller Monat und Jahr
             Datum ();             // Default-Datum: Heute
             Datum (const char*);  // Datum in String-Darstellung
```

- Verwendung von Default-Parametern erlaubt, Anzahl der Konstruktoren zu begrenzen

```
class Datum ...
    public       Datum (int T=0, int M=0, int J=0); ...

Datum::Datum (int T, int M, int J) {
            Tag   = T ? T : heute.Tag;
            Monat = M ? M : heute.Monat;
            Jahr  = J ? J : heute.Jahr;
      // Anweisungen, um Gültigkeit des Datums zu überprüfen
}
```

Klassen (3)

- Destruktoren (z.B. `~Datum`): Durchführung abschließender Aufräumarbeiten wie dynamische Speicherfreigabe, Aktualisierung von statistischen Werten etc.

- Objekte einer Klasse können automatisch, statisch oder dynamisch erzeugt werden
  - lokale Variablen eines Blocks: Konstruktor-Aufruf bei jedem Blockeintritt, wenn Definittion durchlaufen wird; Destruktoren werden bei Blockaustritt in umgekehrter Reihenfolge ausgeführt
  - statische Objekte: Konstrukturausführung gemäß Definitionsreihenfolge; Destruktorausführung am Programmende in umgekehrter Reihenfolge
  - dynamische Objekte: explizite Konstruktor- und Destruktor-Aufrufe (bei `new` bzw. `delete`)

```
Datum today;
void f (int i) {
   Datum d (i);
   //   •••
}
int main () {
   Datum* dp1 = new (11, 11, 95);
   Datum* dp2 = today;
   delete dp1;
   f (20);
}
```


Klassen (4)

- Komponenten können selbst Klassenobjekte sein: Bildung komplexer Objekte
  - im Konstruktor für umfassendes Objekt kann Konstruktor für Komponentenobjekte explizit aufgerufen werden

```
class Termin {
        Datum tag;
        char*Zweck; /* ... */
 public    Termin (int, int); /* ... */ };

Termin::Termin (int T, int M): tag (T, M)
        { /* ... */ }
```

  - Alternative: Verwendung von Pointern auf Klassenobjekte

```
class Termin {
        Datum* tag; /* ... */ };

Termin::Termin (int T, int M) { tag = new Datum (T, M); /* ... */ }
```

- Felder von Klassenobjekten können direkt initialisiert werden:

```
Datum DFeld [] = {1, "20.11.95", Datum(1,1,95), Datum ());
```

Klassen (5)

- *Konstante Elementfunktionen:* Objekt, für das Aufruf erfolgt, darf nicht geändert werden
  - Angabe des Schlüsselworts `const` nach Parameterliste
  - für konstante Objekte dürfen nur konstante Zugriffsfunktionen aufgerufen werden

```
class K {        int m;
      public:    int readme () const { return m;}
                 void writeme (int i) {m = i;} }
```

- Funktionen, die direkt in Klassendeklaration definiert werden, werden als Inline-Funktionen realisiert
  - sehr geringe Ausführungskosten
  - erlaubt effiziente Kapselung von Datenelementen durch Bereitstellung von Reader/Writer-Funktionen

- Selbst-Referenzierung von Objekten über Pointer `this`
  - für jedes Objekt der Klasse K existiert ein solcher Pointer mit impliziter Deklaration
    `const K* this;`
  - wird für jeden Aufruf einer Elementfunktion implizit als zusätzlicher Parameter übergeben
  - Verwendung von `this` innerhalb von Elementfunktionen i.a. nicht erforderlich
    (Ausnahmen: direkte Pointer-Nutzung, z.B. bei verketteten Listen von Klassenobjekten)

Friends

- Klasse X kann Zugriffskapselung auf ihre Elemente für “befreundete” Funktionen, die nicht zu X gehören, selektiv aufheben (friends)

- Beispiel: Multiplikation von Matrizen und Vektoren

```
class Matrix;
class Vektor {double v [5]; // ...
             friend Vektor matXvek (const Matrix&, const Vektor&);
        }
class Matrix {Vektor m [5]; // ...
             friend Vektor matXvek (const Matrix&, const Vektor&);
        }
Vektor matXvek (const Matrix& m, const Vektor& v); {
        Vektor r;
        for (int i=0; i < 4; i++) {        // r [i] = m[i] * v;
              r.v[i]= 0;
              for (int j=0; j < 4; j++) r.v[i] += m.m[i].v[j] * v.v[j]; }
        return r;
   }
```

- Es kann auch eine ganze Klasse Y als “Freund” erklärt werden (Zugriff auf X-Komponenten für alle Elementfunktionen von Y)

```
class X { friend class Y;
             ...             }
```

Statische Klassenvariablen

- Repräsentation gemeinsamer Daten für alle Objekte einer Klasse

```
class Queue
 { // ...
  private:
     static int AnzQueues;
 };
```

```
class Sparbuch
 { // ...
  private:
            static double Zinssatz;
 };
```

- Klasse enthält nur Deklaration, separate Definition und Initialisierung (Default: 0) erforderlich

```
int Queue::AnzQueues;
```

```
double Sparbuch::Zinssatz = 0.035;
```

- auch Elementfunktionen können `static` sein -> Klassenfunktionen

Copy-Konstruktor

- C++ verwendet standardmäßig “flaches” Kopieren
- Probleme bei Klassen/Objekten mit Pointer-Datenelementen

```
 class QElement {  friend class Queue;
                       int     Wert;
                       QElement* next; };

class Queue
 {   QElement*     first;
        int          AnzElemente;
 public:   Queue ();
        void EnQueue (int);
        int DeQueue ();
        int IsEmpty ();        };
```

```
Queue Q1;
Q1.EnQueue (5);
Q2.EnQueue (7);
Queue Q2 = Q1;
int i = Q1.DeQueue ();
```

- tiefes Kopieren kann über einen Copy-Konstruktor realisiert werden
  - hat einen Referenzparameter vom Typ der betreffenden Klasse
  - optional: weitere Parameter mit Standardargumenten

mögliche Prototypen für Queue:

```
Queue::Queue (Queue&)
Queue::Queue (const Queue&)
Queue::Queue (Queue&, int=0)
```

Copy-Konstruktor (2)

- Beispiel

```
Queue::Queue (const Queue& q); {

    first = 0;
    AnzElemente = 0;
    QElement* qp = q.first;
    while (qp != 0)  {
       EnQueue (qp -> Wert);
       qp = qp -> next; }
}
```

- Aufruf des Copy-Konstruktors
  - bei Objektdefinitionen mit Initialisierung über anderes Objekt, z.B.
    `Queue Q2 = Q1` oder `Queue Q2(Q1)`
  - bei Übergabe von Klassenobjekten als Funktionsargumente
  - bei Rückgabe eines Klassenobjektes als Funktionsergebnis, z.B.
    `Queue Vereinigung (const Queue& q1, const Queue& q2);`

- ist kein Copy-Konstruktor definiert, erfolgt Verwendung des Default-Copy-Konstruktors (flaches Kopieren)

Überladen von Operatoren

- nahezu alle Operatoren in C++ können zur Anwendung auf Klassenobjekte (benutzerdefinierte Datentypen) überladen werden

- Definition einer zugehörigen Funktion

```
<Ergebnistyp> operator<op> (...)
```

<op> ist durch das Symbol für den zu überschreibenden Operator (+, - *, /, <<, >>, ...) zu ersetzen

- Beispiel

```
class complex {     double re, im;
public:
  complex (double r, double i) {re=r; im=i;}
  complex operator+ (complex);
  complex operator* (complex);
      };
```

```
complex a = complex (1.2, 2.1);
complex b = complex (2.3, 8.9);
complex c = b;

a = b + c;
b = b + c * a;
c = a * b + complex (1.0, 4.9);
```

- Beschränkungen
  - es können keine neuen Operatoren definiert werden
  - Stelligkeit, Assoziativität und Priorität bleiben ungeändert

Überladen von Operatoren (2)

- binäre Operatoren können definiert werden
  - als Elementfunktion einer Klasse mit einem Parameter oder
  - als globale Funktion mit zwei Parametern (Funktion ist häufig “friend” der betreffenden Klasse)
- unäre Operatoren können definiert werden
  - als Elementfunktion einer Klasse ohne Parameter oder
  - als globale Funktion mit einem Parameter (Funktion ist häufig “friend” der betreffenden Klasse)
- Beispiele

```
class K {
      // Elementfunktionen
  K operator& (K); // binäres &
  K operator++ ();   // Präfix-Inkrement
  K operator++ (int); // Postfix-Inkrement
  K operator- (K, K); // Fehler
  K operator/ ();       // Fehler
      };
```

```
// globale Funktionen (oft friends)

K operator- (K); // Präfix-Minus
K operator- (K,K); // binäres Minus
K operator-- (K&,int);
             // Postfix-Dekrement
K operator+();      // Fehler
K operator+ (K, K, K); // Fehler

Vektor operator* (Matrix&, Vektor&);
```


Überladen von Operatoren (3)

- Zuweisung (=) ist auch für Klassenobjekte vordefiniert: flaches Kopieren

```
class string { public: char* p; int Laenge;
      string (int l) { p = new char [Laenge=l]; }
      ~string () { delete [] p; }
  };
```

```
void f () {
      string s1 (10);
      string s2 (20);

      s1 = s2;
}
```

- Überladen ermöglicht Tiefes Kopieren

```
class string { public:
      // ... wie zuvor
  string& operator= (const string&)
  string (const string&);
};
```

```
string& string::operator= (const string& a)
{     if (this != a) { // s=s ausschließen
         delete []p;
         p = new char [Laenge=a.Laenge];
         strcpy (p, a.p); }
      return *this;
}
string::string (const string& a); {
      p = new char [Laenge=a.Laenge];
      strcpy (p, a.p);
}
```

- Initialisierung und Zuweisung sind unterschiedliche Operationen !

Abgeleitete Klassen

- *abgeleitete Klasse* erweitert *Basisklasse* um eigene Komponenten
- Vererbung erspart Duplizierung der Strukturbeschreibung sowie der Funktionsimplementierung

```
class Konto { const long int Ktonr;
              char*     KtoInhaber;
              double    HabenZins, KtoStand;
  public:     Konto (long int Nr, char* Inh, double HZ, double = 0.0);
              void Info () const;
              void Einzahlung (double); // weitere Funktionen ...
};

class GiroKto: public Konto { double KreditLimit, SollZins;
  public:     GiroKto (long int, char*, double, double, double, double);
              void GiroInfo () const; // weitere Funktionen ...
  }
class Sparbuch: public Konto { date Faelligkeit; // date sei definiert
  public:     Sparbuch (long int, char*, double, double, date);
              void SparInfo () const; // ....
  }
```


Abgeleitete Klassen (2)

- Konstruktoren der Basisklasse werden nicht vererbt
  - jedoch Aufbau eines Objekts beginnend bei Basisklasse durch impliziten Aufruf des Default-Konstruktors (sofern vorhanden)
  - oder expliziter Aufruf von Basisklassen-Konstruktoren

```
Sparbuch::Sparbuch (long int Nr, char* Inh, double HZ,double KS, date d):
                          Konto (Nr, Inh, HZ, KS)
                 {Faelligkeit=d; }
```

- Gleiche Komponenten-/Funktionsbezeichnungen in abgeleiter Klasse führen zu Überdeckung
  - Funktionen werden auch bei abweichender Parameterliste überdeckt
  - Zugriff auf verdeckte Namen über Geltungsbereichsoperator :: (explizite Klassenangabe)

```
class X {
  public:     int i;
              void f ();
}
class Y: public X {
  public:     int i;
              void f (char);
  }
```

```
Y y;

++y.i;
++y.X::i;

y.f("z");
y.f();         // Fehler!
y.X::f();      // o.k.
```

Virtuelle Funktionen

- Überschreiben von Funktionen mit dynamischer Funktionsauswahl erfordert Verwendung von *virtuellen Funktionen*
  - Funktion muß in der Klasse, wo sie zuerst deklariert wird, als `virtual` gekennzeichnet werden
  - Überschreiben mit dynamischem Binden erfordert Verwendung der gleichen Parameterliste (andernfalls Überdeckung)
  - dynamisches Binden erfolgt nur für Zeiger und Referenzen

```
class Konto { // ...
  public:     virtual void Info () const;
};

class GiroKto: public Konto { // ...
             void Info () const;
  }
class Sparbuch: public Konto { // ...
             void Info () const;
  }
```

```
Konto* KFeld [1000] = { 0 };
Konto* k;
KFeld [0] = new GiroKto (...);
KFeld [1] = new Sparbuch (...);
KFeld [2] = new GiroKto (...);
// ...
for int i = 0; i < 1000; i++) {
      k = KFeld [i];
      if (k != 0) k -> Info(); }
  // ...
```

- bei heterogenen Objektkollektionen keine Fallunterscheidung erforderlich, um “richtige” Funktion auszuwählen (-> bessere Erweiterbarkeit)

Abstrakte Klassen

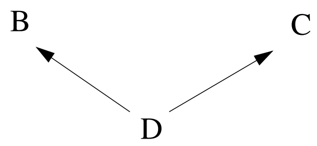
- Abstrakte Klassen
  - Bereitstellung einer gemeinsamen Schnittstelle für abgeleitete Klassen
  - keine Instanzenbildung möglich
- Abstrakte Klasse liegt vor, sobald sie eine *rein virtuelle Funktion* (purely virtual function) hat
  - Initialisierung mit `= 0`
  - keine Implementierung
  - muß in abgeleiteten Klassen überschrieben werden

```
class Figur {  // ...
 public:      virtual double Flaeche() = 0;
              virtual void Zeichnen () = 0;
};

class Polygon: public Figur { // ...
 public:      double Flaeche ();         // ...
}
class Dreieck: public Polygon { // ...
 public:      void Zeichnen (); // ...
}
```

- abgeleitete Klasse ist auch abstrakt, wenn eine rein virtuelle Funktion nicht überschrieben wird
- abstrakte Klassen enthalten oft weder Konstruktor noch Datenelemente

Mehrfachvererbung

- erleichtert Wiederverwendung -> Funktionen mehrerer Basisklassen können genutzt werden



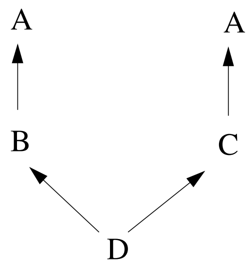

```
class B {      // ...
 public:      virtual void f (); { }
};

class C {      // ...
 public:      virtual void f (); { }
   };

class D: public B, public C { // ...

   };
```

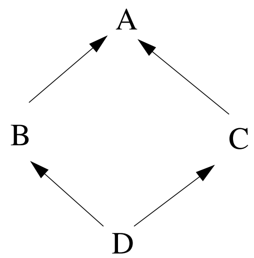
- Auflösung von Namenskonflikten
  - explizite Klassenangabe beim Funktionsaufruf, z.B. `o.B::f();`
  - Overriding in abgeleiteter Klasse

Mehrfachvererbung (2)

- wird Klasse D mehrfach (indirekt) von derselben Basisklasse A abgeleitet, sind die A-Komponenten mehrfach in D-Objekten repräsentiert
- Ausnahme: Verwendung virtueller Basisklassen

virtuell abgeleitete Klasse D teilt sich Daten der virtuellen Basisklasse A mit anderen Klassen, die von derselben Basis A virtuell abgeleitet sind






```
class A { /*  ...  */  };
class B: virtual public A { /* ...*/ };
class C: virtual public A { /* ...*/ };
class D: virtual public B,
         virtual public C { /* ... */ };
```

Zugriffskontrolle

- Zugriffsrechte für Klassenmitglieder (Datenelemente und Funktionen):
  - private: nur nutzbar für Elementfunktionen und Friends der Klasse (Default)
  - protected: nutzbar für Elementfunktionen und Friends der Klasse sowie Elementfunktionen und Friends von abgeleiteten Klassen
  - public: für alle Funktionen nutzbar
- Zugriffsspezifikation für Basisklasse B einer abgeleiteten Klasse A:
  - public: vollständige Vererbung der public- und protected-Elemente von B an A (is-a-Semantik)
  - protected: public- und protected-Elemente von B werden so nach A vererbt, als seien sie in A als protected-Elemente deklariert worden
  - private: public- und protected-Elemente von B werden so nach A vererbt, als seien sie in A als private-Elemente deklariert worden

  private-Elemente von B in A nicht zugreifbar (außer für friend-Funktionen)
- Zugriffsrechte für von B geerbte Elemente in A:

	Vererbungsart		
Zugriffsrecht in B	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Zugriffskontrolle: Beispiel

```
class X {     // ...
      public:  int a;
};

class Y1: public X { };
class Y2: protected X {
      void f2 (); };
class Y3: private X {
      void f3 (); };
```

```
X x; Y1 y1;
x= y1;
y1.a = 7;

Y2 y2;
x = y2;
y2.a = 8;

Y3 y3;
x = y3;
y3.a = 9;
```

```
void Y2::f2 () {
    X x; Y1 y1;
    x= y1;
    y1.a = 7;

    Y2 y2;
    x = y2;
    y2.a = 8;

    Y3 y3;
    x = y3;
    y3.a = 9;
}
```

```
void Y3::f3 () {
    X x; Y1 y1;
    x= y1;
    y1.a = 7;

    Y2 y2;
    x = y2;
    y2.a = 8;

    Y3 y3;
    x = y3;
    y3.a = 9;
}
```

Templates

- Realisierung von
  - generischen Klassen, z.B. zur Implementierung von Kollektions- bzw. Container-Typen (Listen, Mengen, ...) für beliebige Komponententypen
  - generischen Funktionen (z.B. Sortierfunktionen)

- Beispiel: Stack-Template

```
template <class T> class STACK
 { public:
    STACK (const int n); {
            Bottom = Top = new T [MaxAnzahl=n]; }
    ~STACK (); {  delete [] Bottom; }
    void Push(T a) {*Top++ = a;}
    T Pop (); { return *--Top;}
    int Anzahl () const {return Top - Bottom;}

private     T* Bottom;
            T* Top;
            int MaxAnzahl}
```

```
STACK<int> is (100);
STACK<double> ds (100);
STACK<Konto> ks (400);
is.Push (5);
ds.Push (4.9);
ks.Push (new Girokonto);
cout << ks.Anzahl() << endl;
```

- Alternatives Stack-Template:

```
template <class T=int, const int n=100> class Stack { /*...*/ }
```


Listen-Template

- Realisierung einer generischen verketteten Liste

```
template <class E> class ListenElement
{friend class   Liste<E>;
 private:    E* info;
             ListenElement* next;
     ListenElement (E* p, ListenElement* l=0)
             { info = p; next = l;}
     ~ListenElement () { delete info; } };

template <class E> class Liste { public:
     Liste () {  current = first = 0; }
     ~Liste () { }
     void Add (E* p)
      {first = new ListenElement<E> (p, first);}
     void Remove () { }
     /* Iterator-Funktionen */
     void LReset () { current = first; }
     E* getNext() {E* help = 0;
        if (!LEnde() ) {help = current -> info;
                 current=current->next; }
        return help; }
     int LEnde () { return ! current; }
 private:    ListenElement<E> *first, *current;
};
```

```
Liste<Konto> kl;
Liste<Polygon> pl;

for (int i=1;i<8;i++) {
  pl.Add (new Rechteck (i,2*i);
  pl.Add (new Quadrat(i/2); }
pl.LReset();
while (! pl.LEnde()) {
  cout << pl.getNext()->Flaeche()
       << endl; }
```

Listen-Template (2)

- Verkettete Liste mit separaten Iteratoren

```
template <class E> class ListenElement
{ friend class   Liste<E>;
  friend class   Iterator<E>;
  private:    // Rest wie vorher
};
template <class E> class Liste { public:
      Liste () { first = 0; }
      ~Liste () { }
      void Add (E* p)
       {first = new ListenElement<E> (p, first);}
      void Remove () { }
  private:   ListenElement<E> *first, *current;
};
template <class E> class Iterator { public:
  Iterator (Liste<E>& l)     // Konstruktor
{  current = l.first; AktuelleListe = &l;}
  void LReset () {
      current = AktuelleListe -> first; }
  E* getNext() {// wie vorher
      }
  int LEnde () { return ! current; }
  private:   ListenElement<E>* current;
             Liste<E>* AktuelleListe;}
```

```
Liste<Polygon> PListe;
Iterator<Polygon> pi1(PListe),
                  pi2(PListe);
for (int i=1;i<8;i++) {
  PListe.Add (new Quadrat(i/2);
  // ...
}
pi1.LReset();
while (! pi1.LEnde()) {
  cout << pi1.getNext()->Flaeche()
       << endl; }
```


Zusammenfassung: C++

- generelle Merkmale von C++
  - hybride Programmiersprache: imperativ + objekt-orientiert
  - aufwärtskompatibel zu C
  - weitgehend statische Typisierung
  - weite Verbreitung
  - Standardisierung
  - hohe Effizienz
  - großer Sprachumfang
- umfassende Unterstützung objektorientierter Konzepte
  - Klassenkonzept mit statischen und dynamischen Objekten
  - flexibles Konstruktor- und Destruktorkonzept
  - einfache Vererbung sowie Mehrfachvererbung
  - abstrakte Klassen
  - Polymorphismus: Überladen, Overriding, Templates
  - flexible Zugriffskontrolle
- keine automatische Speicherbereinigung

Smalltalk*

- Smalltalk hat historisch der Objektorientierung zum Durchbruch verholfen und gewinnt industriell stark an Bedeutung.
- kleiner Sprachumfang mit reiner objektorientierter Semantik; gewöhnungsbedürftige Syntax
- Smalltalk-Geschichte
  - 1972 Objects, Messages, Classes
  - 1974 erweiterbare Klassen
  - 1976 Vererbung, Entwicklung von interaktiven Werkzeugen
  - 1978 Implementierung effizienter und portierbarer Interpreter
  - 1981 Smalltalk-80, Referenzimplementierungen bei Apple, Berkeley U., HP, Tektronix
  - 1983 Goldberg, Robson: The Language and its Implementation
  - 1984 Goldberg: The Interactive Programming Environment
  - 1986 Durchbruch bei der Implementierung effizienter Interpreter
  - 1990 Integration in das look-and-feel üblicher Fenstersysteme;
    arbeitsteilige Software-Entwicklung, Objektorientierte Datenbanken
  - 1993 Verteilte Anwendungen, CORBA

* http://st-www.cs.uiuc.edu/archive.html

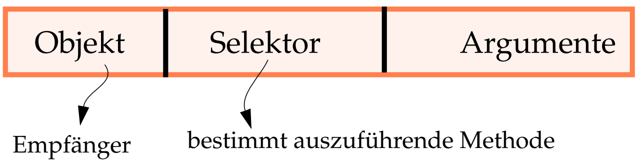
Smalltalk (2)

- Alles ist ein Objekt, auch einfache Werte, Konstanten, Klassen und sogar Anweisungsfolgen
- keine Typisierung -> Konsistenzprüfungen sind statisch nicht möglich
- umfassende dynamische Funktionsprüfung
- strikte Kapselung des Objektzustands
- i.a. keine Mehrfachvererbung
- Entwicklungsumgebung:
  - vorbildlicher Satz interaktiver Entwicklungswerkzeuge
  - Browser
  - Software-Bibliothek mit hoher technischer Qualität
  - mehr als 300 Klassen und 6000 Methoden
- auch unvollständige Programme können ausprobiert werden.

Smalltalk: Grundelemente

- Literale: konstante Objekte
  - Zahlen `7, -5, 4/11, 4.87E-5`
  - Zeichen `$A, $7, $*`
  - Strings `'Smalltalk'`
  - Symbole (nicht änderbare Zeichenketten) `vier`
  - Felder `#(1 2 3)`
- Variablen: Behälter bzw. symbolischer Name für Referenzen auf Objekte
  - Initialwert (Objekt): `nil`
  - Referenzsemantik für Zuweisungen
- private Variable
  - beginnen mit Kleinbuchstabe
  - Instanzvariablen und temporäre Variablen (| ... |)
- gemeinsame Variable
  - beginnen mit Großbuchstabe
  - Klassenvariable, globale Variable, Pool-Variable
- Pseudovariablen (nicht änderbar): `nil, true, false, self, super`

Smalltalk: Nachrichten



- Sender bestimmt 'was' zu tun ist (Selektor), Empfänger das 'wie'
- Nachricht bewirkt Zustandsänderung beim Empfänger oder/und Rückgabe einer Antwortnachricht
- ***Nachrichtentypen:***

a) einstellig (keine Argumente)

```
7 negated
P123 Vorname
sum sqrt
```

b) zweistellig (1 Argument, Selektor: 1 o. 2 Zeichen):

```
8 * 4
myObject = yourObject      (* Gleichheit *)
myObject == yourObject     (* Identität *)
(P1 Gehalt) <= (P2 Gehalt)
```

c) Schlüsselwort-Nachrichten

1 Empfänger, mehrere Keyword (Selektor)-/Argument-Paare

```
liste addFirst: elem
arrayOfStrings at: (2+1) put: 'New York'
```

Nachrichten (2)

- Zusammengesetzte Nachrichten
  - Abarbeitung von links nach rechts

```
'vier' size - #(1 2 3) size
17 + 4 * 3
```

- Sequenzen von Nachrichten: Trennung durch "."

```
liste addFirst:$a.
liste addFirst:$b.
liste addFirst:$c
```

  - entspricht der kaskadierenden Nachricht

```
liste addFirst:$a; addFirst:$b; addFirst:$c
```

- Nachrichtenblöcke
  - Klammerung einer Nachrichtensequenz mit [...]
  - Ausführung erfolgt erst durch Nachricht `value`

```
|iBlock i sumBlock sum|
iBlock := [i <- i + 1].
sumBlock := [sum + (i * i) ].
i := 1.
sum := 0.
```

```
sum := sumBlock value.
iBlock value.
sum := sumBlock value
```


Nachrichten (3)

- Einfache Wiederholungen

```
10 timesRepeat: [sum := sum + sum]
```

- Selektion

allgemeine Form:

```
boolescheNachricht
    ifTrue: trueBlock
    ifFalse: falseBlock
```

Beispiele:

```
(index < 100)
 ifTrue: [liste at:index put:1]
 ifFalse: [liste at:index put:0]
```

```
min := (x < y) ifTrue: [x] ifFalse: [y]
```

- bedingte Wiederholung

allgemeine Form

```
bedingungsBlock
    whileTrue: anweisungsBlock
```

bzw.

```
bedingungsBlock
    whileFalse: anweisungsBlock
```

Beispiele:

```
liste := Array new:100.
i := 1.
[i <= liste size]
    whileTrue:[ liste at:i put:i.
                i := i+1]
```

Klassen und Vererbung

- es gibt nur Objekte, die zu Klassen gehören
- Bildung neuer Klassen durch Erweiterung einer umfangreichen Klassenhierarchie mit Wurzelklasse `Object`
- Unterklasse erbt alle Methoden und Instanzvariablen ihrer Oberklasse (sowie deren Oberklasse(n))

```
Object subclass: #Person
instanceVariableNames: 'name adresse gebDat'

Person subclass: #Professor
instanceVariableNames: 'institut raum telefon'

Object subclass: #Student
instanceVariableNames: 'matnr semester belegteVorlesungen'
```

- Methoden können überschrieben werden
- Klassen sind auch Objekte, nämlich Instanzen von Metaklassen
  - zu jeder Klasse wird implizit eine zugehörige Metaklasse verwaltet
  - jede Metaklasse ist Unterklasse von `MetaClass`

Klassenhierarchie (Auszug)

- Object
  - Magnitude*
    - Date
    - Time
    - Character
    - Number*
      - Integer
        - LargeNegativeInteger
        - LargePositiveInteger
        - SmallInteger
      - Fraction
      - Float
  - File
  - UndefinedObject
    - Boolean
  - DisplayObject
  - Collection*
    - Bag
    - Set
    - SequenceableCollection*
      - LinkedList
      - ArrayCollection*
        - String
          - Symbol
        - Array

Smalltalk: Klassenbeispiel

```
Klasse:             Konto
Oberklasse:         Object
Instanzvariablen: saldo
Klassenmethoden:
      new
         ^super new.
         saldo := 0

Instanzmethoden:
      auszahlen: einBetrag
         saldo := saldo - einBetrag

      einzahlen: einBetrag
         saldo := saldo + einBetrag.

      saldo
         ^saldo.
```

```
Klasse:             Sparbuch
Oberklasse:         Konto
Instanzvariablen: zinssatz
Klassenmethoden:
      new
         ^super new.
         zinssatz := 0.0

Instanzmethoden:
      berechneZinsen: tage
         |zinsen |
         zinsen := saldo * zinssatz / 100.
         zinsen := zinsen * tage / 360.
         saldo := saldo + zinsen.

      zinssatz: einZinssatz
         zinssatz := einZinssatz.
```

```
|einKonto|
einKonto := Konto new.
einKonto einzahlen: 100.
einKonto saldo.

|einSparbuch|
einSparbuch := (Sparbuch new ) zinssatz:3.5.
einSparbuch einzahlen: 2000.
einSparbuch berechneZinsen: 90.
einSparbuch saldo
```


Smalltalk: Beispiel für self/super

```
Klasse:             A
Instanzmethoden:
      ausgabe
         ^ $a
      ergebnis1
         ^ self ausgabe

Klasse:             B
Oberklasse:         A
Instanzmethoden:
      ausgabe
         ^ $b

Klasse:             C
Oberklasse:         B
Instanzmethoden:
      ergebnis2
         ^ self ergebnis1
      ergebnis3
         ^ super ausgabe

Klasse:             D
Oberklasse:         C
Instanzmethoden:
      ausgabe
         ^ $d
```

```
|a, b, c, d|
a := A new.
b := B new.
c := C new.
d := D new
```

Nachricht	Ergebnis
a ausgabe	
a ergebnis1	
b ausgabe	
b ergebnis1	
c ausgabe	
d ergebnis1	
c ergebnis2	
d ergebnis2	
c ergebnis3	
d ergebnis3	
a ergebnis2	

Eiffel*

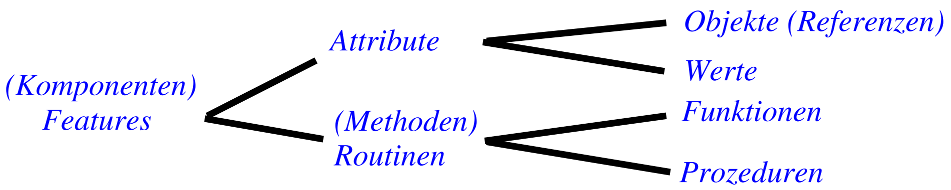
- Eiffel ist eine objektorientierte Sprache mit Typsystem, die von Grund auf neuentwickelt wurde

  "a small, simple language, with strong typing, genericity, multiple inheritance, no global variables, no main program ... and a disciplined exception mechanism" Bertrand Meyer

- wenige Anleihen an imperative Programmiersprachen
  - nur 1 Schleifenkonstrukt (While-Semantik)
  - nur 1 Selektionsanweisung (keine Case-Anweisung)
  - nur Werteparameter für Prozeduren/Funktionen
  - keine explizite Typkonversion

- Unterscheidung zwischen Werten und Objekten
  - Werte: Instanzen von Standard-Datentypen (INTEGER, REAL, CHARACTER, BOOLEAN)
  - Objekte: Instanzen von Klassen (vor- oder benutzerdefiniert)

- Klassen: einheitliche Verwendung von Datenelementen und Methoden (Features)



* B. Meyer: Eiffel: The Language, Prentice Hall, 1992; http://www.eiffel.com/

Eiffel (2)

- Beispiel

```
    CLASS Kugel
    FEATURE {NONE}
       x, y, z: REAL;
    FEATURE {ANY}
       Radius: REAL;
       Volumen: REAL IS
             DO
                    RESULT := PI x 4/3 x Radius^3
             END;
     Verschiebe (Vek1, Vek2, Vek3: INTEGER) IS
             DO
                    x := x + Vek1;
                    y := y + Vek2;
                    z := z + Vek3;
             END;
    END;
...

    k1, k2: Kugel; r1: REAL;
    !!k1;
    k1.Radius := 5.0;
    r1 := k1.Volumen;
    k2 := k1;
    k2.Verschiebe (4, 2, 3);
  ...
```

Eiffel (3)

- Erzeugung und automatische Initialisierung von Objekten (!!-Operator)
  - Komponentenobjekte (Referenzen) werden auf unbelegten Wert (void) gesetzt
  - benutzergesteuerte Initialisierung kann erreicht werden (creation-Klausel)
  - automatische Freispeicherverwaltung
- Zuweisungen
  - Referenzsemantik für Objekte
  - Wertesemantik für Zuweisung von Werten
- vordefinierte Methoden für flaches Kopieren (CLONE) und Test auf flache Gleichheit (EQUAL)

```
o1 := o2;          ->          o1 = o2          (Identität)
o1.CLONE (o2);     ->          o1.EQUAL (o2)    (fl. Gleichheit)
```

- Kapselung: Gruppieren von Features nach Zugriffsrechten

```
FEATURE {ANY}            -- Default: volle Sichtbarkeit
    ...
FEATURE {NONE}           -- private Komponenten
    ...
FEATURE {A}              -- sichtbar nur für Methoden der Klasse A
    ...
```


Eiffel (4)

- Semantik von Abstrakten Datentypen (Klassen) kann spezifiziert werden
  - Vor- und Nachbedingungen für Methoden (REQUIRE bzw. ENSURE)
  - Invarianten für Klasse legen zulässige Zustände für alle Instanzen der Klasse fest (INVARIANT)

```
Pop: eTyp IS
    REQUIRE NOT IsEmpty
    DO ...
```

```
CLASS Kugel
    FEATURE ...
    INVARIANT Radius > 0
END;
```

```
Push (e: eTyp) IS
    DO ...
    ENSURE NOT IsEmpty;
           Top = e
```

- Vorbedingungen sichern Methode vor der nutzenden Umgebung;
  Nachbedingungen sichern Nutzer vor Methodenimplementierung
- benutzerspezifizierte Ausnahmebehandlung für Methode bzw. Klasse (RESCUE-Klausel) bei Verletzung einer Bedingung

Eiffel (5)

- Definition von Klassenhierarchien

```
CLASS Unterklasse
        INHERIT Oberklasse1 [, Oberklasse2, ...]
        FEATURE ...
END;
```

- Klassen- und Methodenbedingungen werden auch vererbt
- Mehrfachvererbung: Behandlung von Namenskonflikten mit RENAME-Klausel (s. 3-45)
- Overriding über REDEFINE-Klausel
  - sogar Attribute können redefiniert werden
  - Ergebnistyp und Argumenttypen von Methoden können in kontrollierter Weise redefiniert werden (s. 3-40)

Eiffel: Klassenbeispiel

```
CLASS Konto
CREATION makeKonto;
FEATURE {NONE}
      saldo : REAL;
FEATURE
makeKonto IS
      DO
         saldo := 0
      END;

einzahlen (betrag: REAL) IS
      DO
         saldo := saldo + betrag
      END;

auszahlen (betrag: REAL) IS
      DO
         require saldo >=  betrag
         saldo := saldo - betrag
      END;

kStand IS
      DO
         RETURN saldo
      END
END -- CLASS Konto
```

```
CLASS Sparbuch CREATION makeSB;
INHERIT Konto
FEATURE {NONE}
 zinssatz: REAL;
FEATURE
makeSB (z: REAL) IS DO
        zinssatz := z END;
berechneZinsen (tage: INTEGER) IS
    DO ... END
      ...
END -- CLASS Sparbuch
```

```
CLASS Test
FEATURE
 make IS
     LOCAL
        einKonto: Konto;
        einSparbuch : Sparbuch
      DO
        !!einKonto.makeKonto;
        !!einSparbuch.makeSB (3.5);
        einKonto.einzahlen(100);
        einSparbuch.einzahlen(200);
        einSparbuch.berechneZinsen(90);
          ...
      END
END -- CLASS Test
```

Eiffel (7)

- Generische Klassen: Verwendung von Typparametern

```
CLASS Stack [eTyp]
FEATURE
     ...
Push (r: eTyp) IS ...
     ...
END -- CLASS Stack
```

```
...
     KugelStack: Stack [Kugel]
...
```

- vordefinierte generische Klassen: Stack, Queue, List, Tree, Array
- nur eingeschränkte Operationen auf Variablen des Parametertyps: :=, =, <>

- Abstrakte Klassen bzw. Spezifikationsklassen
  - mindestens eine Routine ist nur spezifiziert (“aufgeschoben”)
  - anstelle des Rumpfes wird Schlüsselwort DEFERRED angegeben
  - spezifizierte Routine kann Vor- und Nachbedingungen besitzen

```
DEFERRED CLASS Polygon
FEATURE ...

flaeche: REAL IS DEFERRED END;
         ...
END -- CLASS Polygon
```


Vergleich

Merkmal	C++	Smalltalk	Eiffel	Java
Typisierung				
gibt es nur Klassen und Objekte ?				
statische u./o. dynamische Objekte				
automatische Speicherbereinigung				
Zuweisung (Wert- o. Referenzsemantik)				
Mehrfachvererbung				
Generizität				
Kapselung				
gibt es ein Hauptprogramm ?				
Sprachumfang / Komplexität				
Entwicklungseffizienz				
Laufzeiteffizienz				