Objektorientiertes Programmieren
und
C++

Outline

- Entwicklung von OOP Sprachen

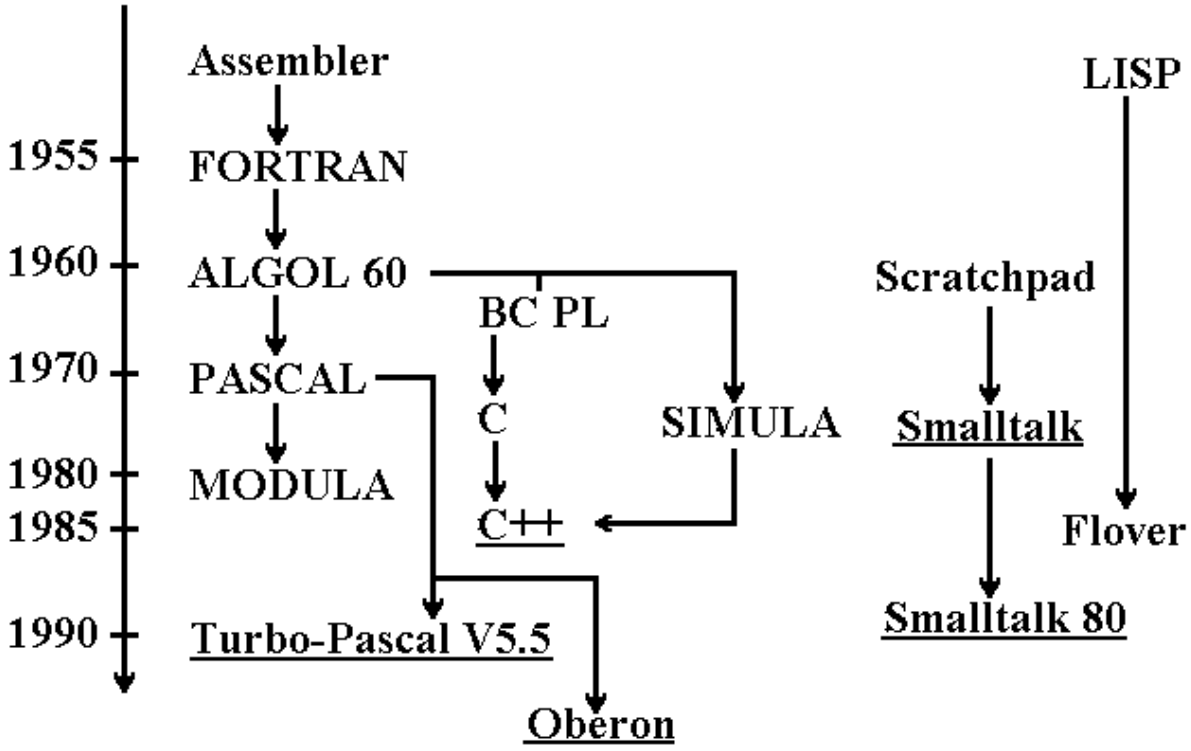
- OOP
  - Wozu OOP?
  - Begriffe
  - Vererbung
  - Lebensbereich von Instanzen
  - Polymorphismus
  - Datenkapselung

- C++
  - Definitionen und Deklarationen
  - Funktionsüberladung
  - Klassendefinitionen/Datenkapselung
  - Vererbung, virtuelle und abstrakte Basisklasse
  - Instanzierung
  - Friends, Operatorüberladung
  - spezielle Variablen und Funktionen

- Beispiel

- Aufgaben

0. Entwicklung von OOP Sprachen



PASCAL	N. Wirth
Oberon	N. Wirth
C	Kernighan, Ritchie
C++	B. Stroustrup
Eiffel	?

Literatur: Bjarne Stroustrup, "Die Sprache C++"
Addison-Wesley

Wirth, "Oberon"

1. **Wozu OOP?**

- **Modulkonzept:**

  - Bildung semantischer Einheiten schon auf Quelltextniveau

  - erste Ansätze der Datenkapselung

  - Aufteilung eines großen Projektes in mehrere kleinere (voneinander unabhängige) Teilprojekte

  - getrenntes Übersetzen und Testen

- **Objektorientiertes Programmieren:**

  - **neue Qualität**

  - Menge von Daten mit zugeordneten Operationen

  - vollständige Datenkapselung

  - Vererbung (Wiederverwendung von schon vorhandenen Implementationen, Weitergabe von Eigenschaften)

2. Begriffe

Objekt: eigenständiges Individuum mit
- charakteristischen Eigenschaften
- (verschiedenen inneren) Zuständen
- Attributen
- Methoden

Beispiel: Kreis *k(m,r)*
- alle Punkte haben den gleichen Abstand zum Mitelpunkt
- Radius
- Konstruktion

Nachricht: Objekte werden durch den Empfang von Nachrichten aktiviert
- Nachricht benutzt eine Methode

Klasse: Zusammenfassung von Objekten mit gleichen Eigenschaften, Methoden und Zuständen
- besitzt Methode zur Erzeugung einer *Instanz*

Objekt: **Instanz** einer Meta-Klasse, die
- das Erzeugen von Klassen ermöglicht
- elementare Klassen beinhaltet
- Objekte verwaltet
- Nachrichtenaustausch ermöglicht

vgl. Turbo-Pascal: *type <ident>=**object** end;*
Beispiel für elementare Klasse: ***INTEGER***

3. Vererbung

Beispiel:

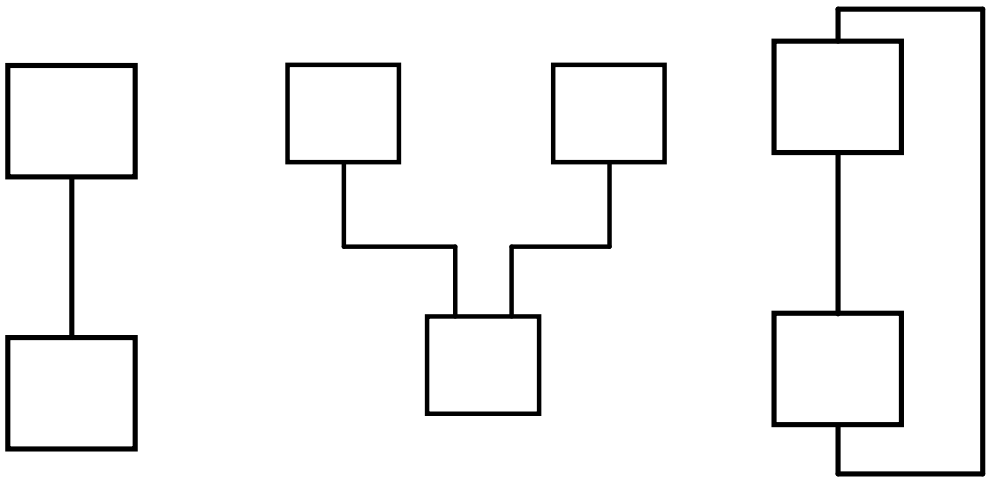
Basisklasse

```
gobject=object
          x,y:real;
          constructor init(x0,y0:real);
          procedure draw;
          destructor done;
        end;
```

abgeleitete Klasse

```
circle=object(gobject)
          x,y:real;
          radius:real;
          constructor init(x0,y0,r0:real);
          procedure draw;
          destructor done;
        end;
```

einfache Vererbung | mehrfache Vererbung | zyklische Vererbung



4. Lebensbereich von Instanzen

- **statische erzeugte Instanz**

```
x:circle;
```

  - nur im Block und erst ab der Stelle an der die Deklaration steht

  - in C++ wird Konstruktoraufruf (Standardkonstruktor) impliziert, Pascal verlangt expliziten Aufruf

- **dynamisch erzeugte Instanz**

PASCAL	C++
x:^gobject;	gobject x;
...	...
x:=new(circle,init(...))	x=new circle(...);

  - *new* allokiert Speicher für ein Instanz vom Typ *circle*, ruft danach den (Standard)Konstruktor auf, und gibt eine Referenz (Zeiger) auf die Instanz zurück

  - die Instanz "lebt" bis zum expliziten Desktruktoraufruf *dispose(x,done)* bzw. in C++ wird durch *delete x* ein Desktruktoraufruf impliziert

5. Polymorphismus

Beispiel:
```
case type(x^) of
    GOBJECT:   gobject(x^).draw;
    CIRCLE:    circle(x^).draw;
end;
```

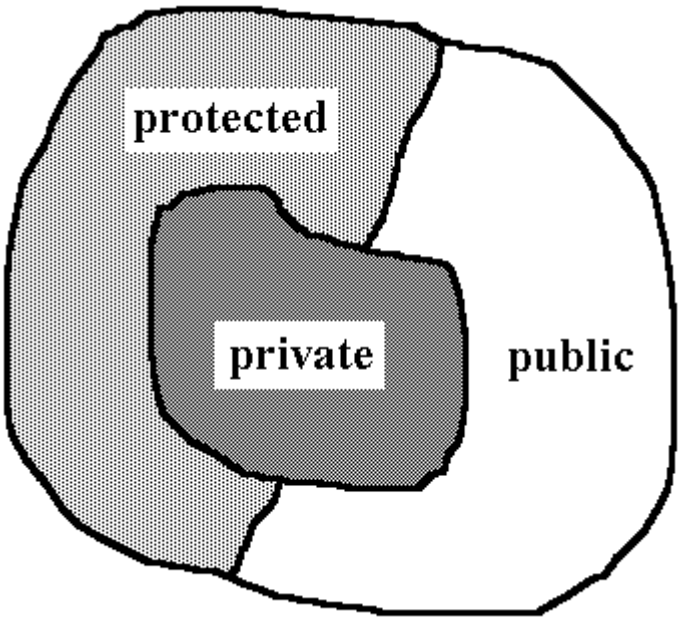
Polymorphismus (späte Bindung):

Bilden von Referenzen auf Objekte, deren Klassenzugehörigkeit erst zur Laufzeit bekannt ist.

⇨ **virtuelle Methoden**

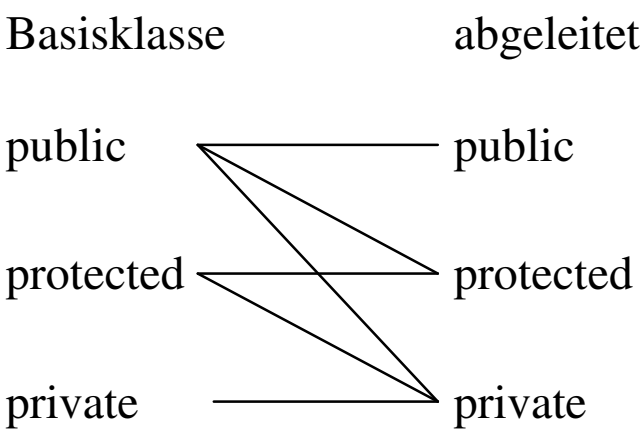
```
gobject=object
            ...
            procedure draw; virtual;
            ...
        end;

circle=object(gobject)
            ...
            procedure draw; virtual;
            ...
        end;

x:^gobject;
...
x:=new(circle,init);
x^.draw;
```

6. Datenkapselung



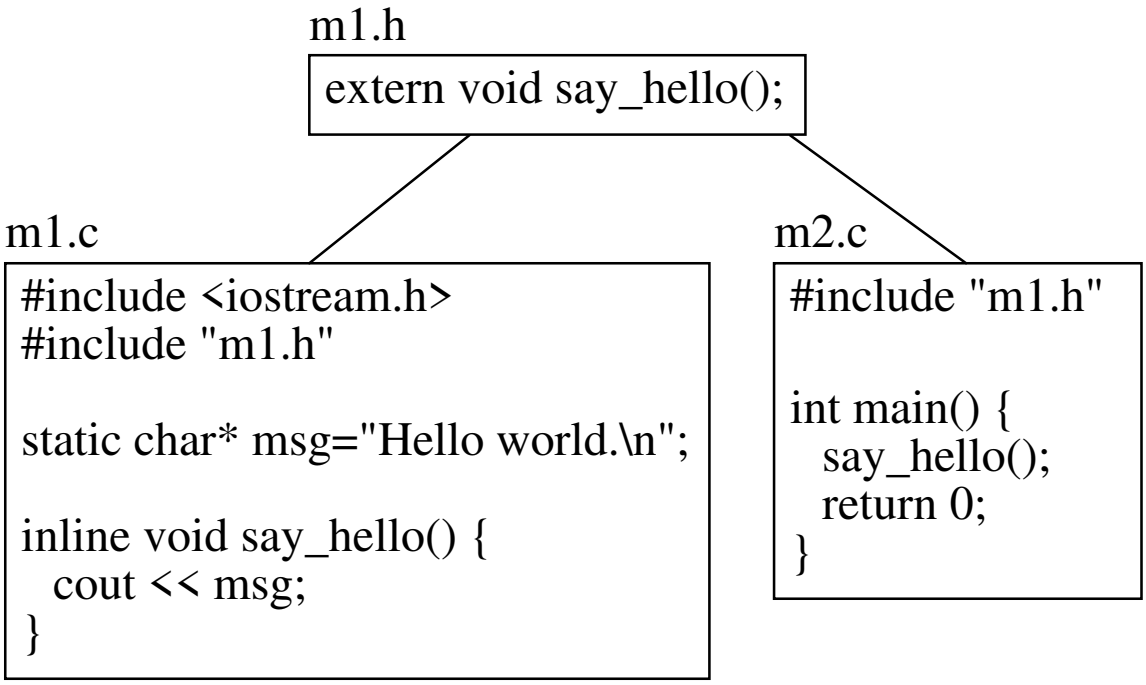
public: Zugriff von überall her
protected: Zugriff nur aus Klassenmethoden, abgeleiteten Klassen, friend Funktionen
private: Zugriff nur aus Klassenfunktionen

Vererbung von Zugriffsrechten:



7. C/C++

Ein C/C++ Programm ist eine Menge von Funktionen, unter denen genau eine den Namen ***main*** trägt. Beispiel:



Datentypen: [unsigned] [long | short] int
float, double
[unsigned] char
void

Zeiger und Felder: <typeid> *
<typeid> []

Strukturen: struct <structname> { <komponenten> }

Typdefinitionen: typedef <definition> <typname>

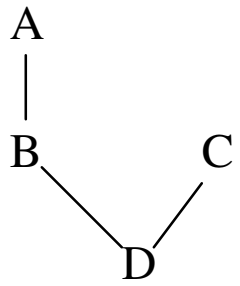
Klassen: class <klassenname> { <komponenten> }

8. Funktionsüberladung

```
int f(int x) { return x+1; }
float f(float x) { return 2*p; }
int *f(char *x) { return strcpy(x+1,"++"); }

main() {
   int x=1; float y=0.3; char *z="CPP";
   cout << f(x) << "\n" << f(y) << "\n" << f(z) << "\n";
}
```

```
2
0.6
C++
```

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




```
D* dptr; f(dptr,dptr)
```

f(A*,A*), f(B*,C*)	f(B*,C*)
f(A*,A*), f(B*,C*), f(C*,C*)	kein Aufruf möglich
f(A*,B*), f(B*,A*), f(C*,C*)	kein Aufruf möglich

Auswahlreihenfolge:

niveau	conversion
1	exact match trivial conversion
2	integral promotion float ☞ double
3	standard conversion class pointer conversion class reference conversion
4	user defined conversion
5	ellipsis conversion

Relation $(t \to t') > (t \to t'')$ (besser als):

- $(t \to t')$ ist kürzer als $(t \to t'')$
- niveau$(t \to t') <$ niveau$(t \to t'')$
- niveau$(t \to t')$ = niveau$(t \to t'')$, $(t \to t')$ enthält bessere Konvertierung des Niveaus 3 als $(t \to t'')$

Auswahlverfahren:

1. $T_i = \{t : \exists f(\ldots, t_i, \ldots) \wedge t = t_i\}$
2. $BM_i = \{f(\ldots, t_i, \ldots) : t_i$ best match für typ_i bzgl. $T_i\}$
3. $BM = \bigcap_{i=1}^{n} BM_i = \{f(TYP_1, \ldots, TYP_n)\}$
4. $\forall f(t_1, \ldots, t_n) \in \bigcup_{i=1}^{n} BM_i \setminus BM$

   $\Rightarrow \exists j \in [1,n] : (typ_j \to TYP_j) > (typ_j \to t_j)$

9. Klassendefinitionen

Klassendeklaration:

```
class classname:public parentclass,
                protected otherclass {
    private:                  // kann weggelassen werden
        int i;                // privates Attribut
        int m1(char *s);      // private Methode
    protected:
        float x;              // geschütztes Attribut
        virtual void m2();    // geschützte virtuelle Methode
    public:
        char *myname;         // öffentliches Attribut
        classname();          // Standardkonstruktor
        int m3();             // öffentliche Methode
        virtual int m4()=0;   // rein virtuelle Methode
        ~classname();         // Destruktor
};
```

Methodendefinitionen:

```
<classname>::<method>(...) { ... }
```

Konstruktor:

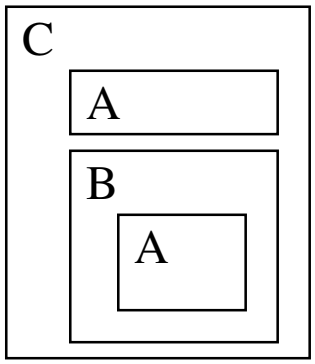
```
classname::classname()
            :parentclass(), otherclass() {
    ...
}
```

10. Vererbung, virtuelle und abstrakte Basisklasse

class::="class" classname ":" base_class_list
base_class::=["virtual"] [access_spec] [class_key] classname

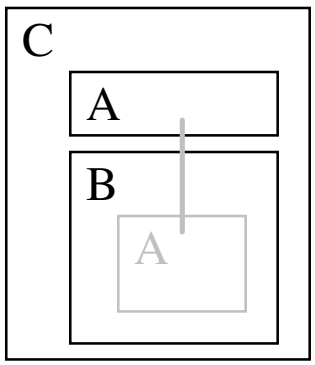
normale Vererbung:

```
class A {};
class B:class A {};
class C:class A,class B {};
```



virtuelle Basisklasse

```
class A {};
class B:virtual class A {};
class C:class A,class B {};
```



abstrakte Basisklasse:

Ein Klasse, die wenigstens eine rein virtuelle Funktion enthält heißt abstrakte (Basis)klasse. Abstrakte Klassen sind nicht instanzierbar, aber es ist möglich Referenzen zu bilden.

Vererbung von Zugriffsrechten:

class B:

public A	protected A	private A	class A { ... };
-	-	-	private
protected	protected	private	protected
public	protected	private	public

11. Konstruktion einer Instanz

1. Konstruktion der Instanzen der direkten virtuellen Basisklassen

2. Konstruktion der Instanzen/Teile der direkten nichtvirtuellen Basisklassen

3. Konstruktion der Instanzvariablen

4. Konstruktor der abgeleiteten Klasse

12. Friends und Operatorüberladung, das Attribut *this*

```
class complex {
        double real,imag;
    public:
        ...
        // unärer Operator
        complex operator-();
        complex &operator=(complex &a);
        // binärer Operator
        friend complex operator+(complex &a,complex &b);
        ...
};
```

Friends können auf alles aus der Klasse, in der sie erklärt sind Bezug nehmen.

```
complex operator+(complex &a,complex &b) {
    return complex(a.real+b.real,a.imag+b.imag);
}
```

Copy-Konstruktor:

```
complex &complex::operator=(complex &a) {
    real=a.real;
    imag=a.imag;

    return *this;
}
```

this: Das Attribut *this* ist eine Referenz auf seine Instanz.

13. Komplettes Beispiel

Implementation eines Restklassenringes modulo n.

```
class Z {
        int n;
        int v;
    public:
        Z(int modul, int v0=0);
        Z &operator=(Z &x);
        Z operator-();
        friend Z operator+(Z &x,Z &y);
        friend Z operator-(Z &x,Z &y);
        friend Z operator*(Z &x,Z &y);
};

Z::Z(int modul, int v0) {
    n=modul;
    v=v0 % n;
}

Z &Z::operator=(Z &x) {
    n=x.n;
    v=x.v;
    return *this;
}

Z operator+(Z &x,Z &y) {
    return Z(x.n,x.v+y.v);
}

Z operator-() {
    return Z(n-v-1,n);
}

Z operator-(Z &x,Z &y) {
    return Z(x.v-y.v,x.n);
}

Z operator*(Z &x,Z &y) {
    return Z(x.v*y.v,x.n);
}
```

14. Aufgaben

1. Entwerfen Sie eine Klasse *complex*, die die Arbeit mit komplexen Zahlen ermöglicht.

2. Entwerfen Sie eine Klasse *vector*, die einen Vektorraum über dem Körper der reellen Maschinenzahlen mit beliebig, aber fest zu wählender Dimension implementiert.

   Hinweis:

   Felder können in C/C++ dynamisch allokiert werden. Benutzen Sie dazu in C++ die Konstrukte

   type " *" ident ";"
   ident "=" "new" type "[" dimension "]" ";"

   um ein eindimensionales Feld der Länge *dimension* zu erzeugen und

   "delete" ident ";"

   um den belegten Speicherplatz wieder freizugeben. Der Zugriff auf ein Feldelement erfolgt mittels

   ident "[" index "]"

3. Entwerfen Sie eine Klassenhierarchie, die den Umgang mit geometrischen Primitiven (etwa Punkt, Gerade, Strahl, Strecke, ...) ermöglicht. Benutzen Sie zum Beispiel die klassischen Definitionen nach Euklid.