Vererbung (inheritance)

- ❑ neue Klassen können durch Ableitung aus ex. Klassen entstehen
- ❑ Ableitung bedeutet “ist ein”
- ❑ Bezeichnungen: Unterklasse / Oberklasse (oder Basisklasse)
- ❑ Erben gewisser Eigenschaften der Basisklasse
- ❑ eine Basisklasse: einfache Vererbung, sonst mehrfache Vererbung
- ❑ Syntax:

```
class Arbeiter : public Person
{
    ...
};
```


Vererbung

- ❑ In C++: Alles was nicht reimplementiert wird, wird vererbt:

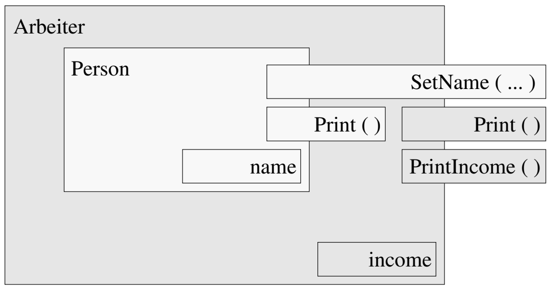
```
class Person { ...
    public:        void Print ( );
                   void SetName ( ... );
};
class Arbeiter : public Person { ...
    public:        void Print ( );
                   void PrintIncome ( );
};
```

verhält sich wie:

```
class Arbeiter {
    public:        void Print ( );          // aus “Arbeiter”
                   void PrintIncome ( );    // aus “Arbeiter”
                   void SetName ( ... );    // aus “Person”
};
```


Vererbung

- ❑ “Durcherben” von Methoden:
  die Schnittstelle wird durch Ableitung vergrößert



Bereichsauflösungs-Operator

- ❑ Oft ist Zugriff der Unterklasse auf reimpl. Oberklassenmethode nötig
- ❑ Verwendung des Bereichsauflösungs-Operators “::”

```
class Arbeiter: public Person {
    void Print ( ) {
            // Print ( );                    // so nicht (Endlosrekursion)
            Person :: Print ( );
            cout << “Gehalt: “ << income << “\n”;
    }
};
...
Arbeiter a;
a . Print ( );
a . Person :: Print ( );
```

Sichtbarkeit bei Vererbung

❑ Unterklasse darf auf **protected** u. **public** Teil der Oberklasse zugreifen

❑ Unterklasse hat keinen Zugriff auf **private** Teil der Oberklasse

❑ von außerhalb ist nur public-Teil zugreifbar

```
class Person {
    public:        void Print ( );
    protected:     char* GetName ( );
    private:       char* name;
};
```

```
class Arbeiter : public Person {
    public:        void PrettyPrint ( ) { Print ( ); ... = GetName ( ); }// ok
                   void PrintName ( ) { cout << name; } // geht nicht, da private
};
```


Konstruktoren

❑ Initialisierung der Oberklasse erfolgt durch Oberklassenkonstruktor

❑ Unterklassenkonstruktor muß Oberklassenkonstruktor aufrufen

❑ Syntax:

```
Arbeiter :: Arbeiter (char* name, int income) :
    Person (name)
{
    ... GetName ( );
}
```

❑ Oberklassenkonstruktor wird vor Unterklassenkonstruktor aufgerufen

Destruktor

❑ Deinitialisierung der Oberklasse muß durch Oberklasse erfolgen

❑ wird automatisch nach Destruktor der Unterklasse aufgerufen

❑ Syntax:

```
Arbeiter :: ~Arbeiter ( )
{
    ...
}
```


Initialisierung eingebetteter Objekte

❑ eingebettete Objekte werden nach der Oberklasse initialisiert

❑ Beispiel:

```
class Arbeiter : public Person
{
    Klasse1 instVar1;
    Klasse2 instVar2;
    Arbeiter (char* name);
}

Arbeiter :: Arbeiter (char* name) :
    Person (name),
    instVar1 (123),
    instVar2 (234)
{ ... }
```

Zeiger auf Objekte

- ❏ Zeiger auf Unterklassen-Objekt kann Zeiger auf Oberklassenobjekt zugewiesen werden
- ❏ Umkehrung gilt nicht
  - erfordert explizite Typkonvertierung
  - ist in manchen Fällen aber unumgänglich
- ❏ allgemeine Regel:
  - spezieller Typ kann einem allgemeinen Typ zugewiesen werden
- ❏ die Schnittstelle legt der Zeigertyp, nicht der Objekttyp fest

Typumwandlung

- ❏ Beispiel:

```
class Person { ... };
class Arbeiter: public Person { ... };
...
Arbeiter* a = new Arbeiter;             // ok
Person* p = new Person;                 // ok
Person* pa = a;                         // ok
Arbeiter* a1 = p;                       // falsch
Arbeiter* a2 = pa;                      // funktioniert nicht
Arbeiter* a3 = (Arbeiter*) pa;          // ok
Arbeiter* a4 = (Arbeiter*) p;           // Fehler
```


virtuelle Methoden

- ❏ Schnittstelle im bisherigen Fall:
  - Zeiger, nicht das Objekt legt Schnittstelle und Semantik fest
- ❏ bei sogenannten *virtuellen* Methoden (dynamischer Fall):
  - Objekt, nicht der Zeiger legt Semantik fest
- ❏ Syntax:

```
class Test
{
    virtual void Print ();
}
```

- ❏ **virtual** muß in der Basisklasse angegeben werden

Polymorphismus

- ❏ Aufruf einer Methode, ohne das Objekt genau zu kennen
- ❏ tatsächliche Methode wird erst zur Laufzeit des Programms bestimmt
- ❏ Zeiger kann auf Objekte verschiedener Klassen zeigen
- ❏ Eindeutigkeit ist gewahrt:
  - Objekt hat immer definierten Typ

virtuelle Methoden

Beispiel:

```
class Person { ...
    public: void Print( ) { ... cout << name << Beruf( ) << ...; }
            virtual char* Beruf( ) { return "unbekannt"; }
};

class Arbeiter: public Person { ...
    public: virtual char* Beruf( ) { return "Arbeiter"; }
};
class Angestellter...
...
Person* p [4] = {  new Arbeiter("Hans"), new Angestellter("Peter"),
                   new Person("Kurt"), new Arbeiter("Tom") };

for (int n = 0; n < 4; n++) p [n] -> Print ( );
```


Mehrfachvererbung

❏ abgeleitete Klasse kann von mehreren Klassen abgeleitet werden

❏ abgeleitete Klasse vereint Verhalten aller Basisklassen

❏ ACHTUNG: Namenskonflikte möglich !

- Lösung: Elemente der Basisklasse mit scope-Operator verwenden

❏ Syntax:

```
class derived : public base1, public base2
{
    ...
}
```


Operator-Overloading

❏ in C++ können Operatoren überladen werden

❏ wird zurückgeführt auf Funktions / Methodenüberladung

❏ Syntax:

- Prototyp: *Returndatentyp* **operator** *op* (*Paramterliste*)

❏ Überladbare Operatoren

+	-	*	/	%	^	&	|	~	!
=	>	<	+=	-=	*=	/=	%=	^=	&=
|=	<<	>>	>>=	<<=	==	!=	<=	>=	&&
||	++	--	,	->*	->	( )	[ ]	new	delete

Globale Operatoren

❏ werden wie Funktion behandelt

❏ Beispiel:

```
class Person { ...
    char* name;
    friend ostream& operator << (ostream& os, const Person& p);
};                                              // "name" zugreifbar
ostream& operator << (ostream& os, const Person& p) {
    os << p.name;
    return os;
}
...
Person p("Hans");
cout << p;                  // Aufruf als Operator
operator << (cout, p);      // Aufruf als "normale" Funktion
```