Strukturerweiterung in C++

❏ neues **struct**-Konzept

- **struct**-Definition ist gleichzeitig ein **typedef**
- lokale Funktionen innerhalb eines **struct**

❏ Beispiel:

```
struct person
{
    int     alter;
    char* name;

    int     Alter ();
    char* Name ();
};
```


Klassen

❏ C++ Klassendeklaration: **class**

- Erweiterung der **struct**-Konstruktion
- enthält Daten- und Methodendeklarationen
- Nachricht bedeutet in C++: Element dereferenzieren

❏ Beispiel:

```
class Person
{
    int     alter;
    char* name;

    int     Alter ();
    char* Name ();
};
```


Schnittstellen von Klassen

❏ Aufteilung eines Objekts in Bereiche:

- **private**: nur Klasse kann Elemente referenzieren
- **public**: jeder kann Elemente referenzieren
- **protected**: wie **private**, Subklassen dürfen jedoch zugreifen

❏ **public**-Bereiche kennzeichnen Schnittstelle nach außen

❏ default-Wert ist **private**

❏ Bereiche dürfen sich beliebig abwechseln

❏ Bereiche dürfen auch mehrfach vorkommen

Schnittstellen in Klassen

❏ Beispiel:

```
class Person
{
private:

    int     alter;          // Instanz-
    char* name;             // Variablen

public:

    int     Alter ();       // Schnittstellen-
    char* Name ();          // Methoden
};
```

Objekterzeugung

❑ syntaktisch äquivalent zu Variablendefinition

❑ statische Erzeugung

```
Person hans;
Person paul;
```

❑ dynamische Erzeugung mit **new**

```
Person *hans;
hans = new Person;
delete hans;
```


Objektzugriffe

❑ Zugriffe außerhalb des Objekts

- private Instanzvariablen sind nicht zugänglich
- private Methoden sind nicht zugänglich
- öffentliche Methoden und Instanzvariablen sind zugänglich

❑ Beispiel:

```
Person hans;
Person *paul = new Person;

hans.Alter ();          // richtig, da Alter() public
hans.alter;             // Fehler, private Inst-var
paul -> Name ();        // richtig, da Name() public
paul -> name;           // Fehler, private Inst-var
```


Klassenfunktionen (Methoden)

❑ Definition innerhalb der Klasse

- Methodenrumpf kommt direkt hinter Deklaration
- Methode wird automatisch **inline**
- Klassen normalerweise in header files (.h)

❑ Definition außerhalb der Klasse

- Innerhalb der Klasse nur Prototyp angegeben
- Methode wird der Klassenname vorangestellt
- Trennung erfolgt durch scope-operator **::**
- Implementierung normalerweise in .C files (.cxx, .C, .cpp)

Klassenfunktionen (Methoden)

❑ Beispiel:

```
class Person
{
    int alter;
    char* name;

    public:
    int Alter () { return alter; }
    char* Name ();
};

char* Person :: Name ()
{
    return name;
}
```

Konstruktoren

- ❏ Klassenmethode
- ❏ Methodenname ist Klassenname
- ❏ kein Returndatentyp (auch nicht void)
- ❏ durch Overloading verschiedene Konstruktoren möglich
- ❏ Konstruktoren normalerweise im **public**-Bereich
- ❏ Zweck: Objekt wird automatisch initialisiert
- ❏ etwas ähnliches wie **init**() ist nicht mehr nötig
- ❏ Compiler stellt sog. „default Konstruktor“ zur Verfügung

Destruktoren

- ❏ Klassenmethode
- ❏ Methodenname ist ~Klassenname
- ❏ kein Returndatentyp (auch nicht void)
- ❏ nur <u>**ein**</u> Destruktor möglich
- ❏ Destruktoren normalerweise im **public**-Bereich
- ❏ Destruktoren haben keine Parameter
- ❏ Zweck: Aufräumarbeiten vor Objektvernichtung
- ❏ Compiler stellt „default Destruktor“ zur Verfügung

Memberobjekte

- ❏ Klassenobjekte innerhalb anderer Klassen

```
class Arbeitsplatz
{
    Person arbeiter;
    ...
};
```

- ❏ Zugriff erfolgt mit **.** oder **->**
- ❏ Probleme:
  - Werden Konstruktoren der Objekte aufgerufen ?
  - Wenn ja, wann werden sie aufgerufen ?
  - Welche Parameter bekommen die Aufrufe ?
  - Wo werden die Konstruktor-Aufrufe platziert ?

Memberobjekte

```
class Person
{
  public:
        Person ();
        ...
};

class Arbeitspl
{
        Person p;
        ...
};

Arbeitspl::Arbeitspl ()
{ ... }
```

```
class Person
{
  public:
        Person (char*);
        ...
};

class Arbeitspl
{
        Person p;
        ...
};

Arbeitspl::Arbeitspl () :
        p (“Hans“)
{ ... }
```

komponenetenweise Initialisierung

❑ Initialisierung eines Objekts mit einem bereits existierenden

```
Person paul (hans);
```

❑ wann wird dieser Mechanismus benötigt ?

- Objekt als Funktionsparameter bei call-by-value
- Rückgabe eines Objekts als Funktionswert
- Initialisierung eines Objekts wie oben

❑ Default-Behandlung:

- initialisiert jede Komponente mit der entspr. des zweiten Objekts
- rekursiver Abstieg

komponenetenweise Initialisierung

❑ benutzerdefiniertes Verhalten:

- Definition eines zusätzlichen Konstruktors

```
Person :: Person (const Person& obj)
```

- es finden keine default-Initialisierungen statt
- gesamtes Kopieren in der Verantwortung des Konstruktors
- auch weitergehende Aktionen möglich
- Beispiel: Puffer-Objekt

Objekte als Feldelemente

❑ Statische Arrays

- ohne Initialisierung
  für alle Elemente wird Standardkonstruktor aufgerufen

```
Person test[4]                // ruft 4mal Person::Person() auf
```

- mit Initialisierung
  spezifizierte Konstrukt. werden auf erste Objekte angewendet
  alle anderen werden mit Standardkonstruktor aufgerufen

```
Person test[4] = {“Fritz“,
                  Person (“Hans“) };
   // test[0] und test[1] : Person::Person (char*)
   // test[2] und test[3] : Person::Person ()
```


Objekte als Feldelemente

❑ Dynamische Arrays

- es wird immer der Standardkonstruktor aufgerufen

```
Person *tabelle;
tabelle = new Person[4];              // 4 mal Standardkonstrukt.
```

❑ Löschen von Arrays

- für alle Elemente wird der Destruktor aufgerufen
- dynamische Arrays müssen mit **delete[]** gelöscht werden

this-Zeiger

- ❏ zeigt innerhalb eines Objekts auf das Objekt selbst
- ❏ wird implizit bei Methodenaufrufen übergeben
- ❏ verhält sich wie: **X *const this;**
- ❏ bei const-Methoden: **const X *const this;**

```
class Person { ...
    char* name;
public:
    void Print ( )   { cout << this -> name; }                    // = name
    void InsertInto (List* l){ l -> Insert (this); }
    void PrettyPrint ( ) { cout << “Daten:”; this -> Print ( ); }   // = Print ( )
};
```


Friends

- ❏ grundsätzlich hat nur Klasse selbst Zugriff auf **private**-Bereich
- ❏ Klasse kann jedoch Methoden oder Klassen zu „friends“ erklären
- ❏ friends haben uneingeschränkten Zugriff auf **private**-Bereich
- ❏ ACHTUNG: umgeht <u>sämtliche</u> Kapselungen der Klasse

Friends

- ❏ Beispiel:

```
class Person
{
    friend void Print (ostream&, const Person&);
    char * Name;
    ...
};

void Print (ostream& o, const Person& p)
{ o << p.Name;
}

void main ( )
{   Person hans;
    Print (cout, hans);
}
```


static-Elemente

- ❏ normal: Jedes Objekt erhält einen eigenen Satz Instanzvariablen
- ❏ Ausnahme: als static deklarierte Instanzvariablen
- ❏ ermöglichen ein sharing auf Klassenebene
- ❏ Initialisierung erfolgt global, außerhalb der Klasse
- ❏ <u>Beispiel</u>:

```
class Test
{   ...
    static int a;
    ...
}

Test :: a = 5;
```