Klassen

- Strukturen in C und in C++
  - Deklaration von Strukturen
  - Funktionen als Bestandteil von Strukturen
  - Datenkapselung: `public`, `private` und `class`

- Konstruktoren und Destruktoren
  - Der Konstruktor
  - Der Destruktor
  - Eine erste Anwendung
  - Konstruktoren mit Argumenten

- Die Reihenfolge der Erzeugung und Zerstörung von Objekten

- `const`-Elementfunktionen und `const`-Objekte

- Klassen und Dynamische Speicherverwaltung
  - `new`, `delete` und Zeiger auf Objekte
  - Dynamisch allokierte Arrays

- `inline`-Elementfunktionen

- Objekte in Objekten: Komposition
  - Initialisierungslisten

- `friend`-Funktionen und `friend`-Klassen

Strukturen in C und in C++

Deklaration von Strukturen

Beispiel: Strukturdeklaration in **C**

```
struct SomeStruct
{
  int a;
  double d;
  char string[80];
};
```

Definition einer Variablen vom Typ `SomeStruct` in **C**:

```
struct SomeStruct what;
```

Oder aber:

```
typedef struct
{
  int a;
  double d;
  char string[80];
} SomeStruct;
...
SomeStruct what;
```

In **C++** ist das `typedef` nicht mehr notwendig. Nach der Deklaration kann `SomeStruct` wie ein Typname verwendet werden.

Strukturen in C und in C++

Funktionen als Bestandteil von Strukturen

In C++ ist es möglich, Funktionen als Bestandteil von Strukturen zu definieren.

```
struct Point          // Definition eines Punktes
{                     // z.B. auf einem Bildschirm
  int x, y;           // die x- und y-Koordinaten
  void draw(void);    // Funktion zum Zeichnen
};
```

Die Funktion `draw()` wird hier lediglich *deklariert*.

Die Struktur `Point` ist `2*sizeof(int)` groß.
Die Größe der Struktur wird durch die Funktion nicht beeinflusst.

Verwendung der Struktur `Point` z.B. folgendermaßen:

```
Point a, b;     // zwei Punkte auf dem Bildschirm

a.x = 0;        // definiere den ersten Punkt
a.y = 10;
a.draw();       // und zeichne ihn

b = a;          // kopiere a nach b
b.y = 20;       // ändere y-Koordinate
b.draw();       // und zeichne b
```

Strukturen in C und in C++

Funktionen als Bestandteil von Strukturen

werden auch *Elementfunktionen* oder *Methoden* genannt.

Beispiel:

```
struct Person
{
  char name[80], address[80];
  void print(void);   // hier nur Deklaration
};
```

Bei der Definition der Elementfunktion `print()` werden der Strukturname und der Bereichsoperator `::` angegeben.

```
void Person::print()
{
  cout << "Name:    " << name << endl;
  cout << "Address: " << address << endl;
}
```

Verwendung dieser Struktur z.B. folgendermaßen

```
Person p;

strcpy(p.name, "Otto");
strcpy(p.address, "Kanalstr. 67");
p.print();
```

Strukturen in C und in C++

Funktionen als Bestandteil von Strukturen

In C benötigen Funktionen zur Bearbeitung von Strukturen ein explizites Argument, das die Struktur bezeichnet.

```
/* C-Definition einer Struktur PERSON_ */
typedef struct
{
  char
    name[80],
    address[80];
} PERSON_;

/* Deklaration einiger Funktionen zur */
/* Manipulation von PERSON_ Strukturen */

/* Initialisierung der Datenfelder */
/* mit einem Namen und einer Adresse */
extern void initialize(PERSON_ *p,
                       char const *nm,
                       char const *adr);

/* Informationen ausgeben */
extern void print(PERSON_ const *p);

/* usw. ... */
```

Aufruf z.B. der Funktion `initialize()`:

```
PERSON_ x;

initialize(&x, "some name", "some address");
```

Strukturen in C und in C++

Funktionen als Bestandteil von Strukturen

In C++ sind Elementfunktionen möglich. Das `struct`-Argument ist in C++ implizit.

```
class Person
{
  public:
    void initialize(char const *nm,
                    char const *adr);
    void print(void);
    // usw. ...
  private:
    char
      name[80],
      address[80];
};
```

Aus dem C-Funktionsaufruf

```
PERSON_ x;

initialize(&x, "some name", "some address");
```

wird in C++

```
Person x;

x.initialize("some name", "some address");
```

Strukturen in C und in C++

Datenkapselung: `public`, `private` und `class`

zwei Schlüsselworte im Zusammenhang mit dem Zugriff auf Elementdaten und -funktionen: `public` und `private`

- `public`: alle folgenden Strukturelemente unterliegen keinerlei Zugriffsbeschränkung

- `private`: alle folgenden Strukturelemente sind ausschließlich innerhalb der Struktur zugreifbar

Voreinstellung für Strukturen: alle Elemente sind `public`

```
struct Person
{
  public:
    void
      setname (char const *n),
      setaddress (char const *a),
      print (void);
    char const
      *getname (void),
      *getaddress (void);
  private:
    char
      name [80],
      address [80];
};
```

Strukturen in C und in C++

Datenkapselung: `public`, `private` und `class`

Auf die Elementdaten `name` und `address` dürfen nur die Elementfunktionen zugreifen, die in `Person` definiert sind.

```
Person x;

x.setname("Hans");
// ok, setname() ist public

strcpy(x.name, "Lisa");
// Fehler, name ist private
```

Beispiele für Elementfunktionen der Struktur `Person`:

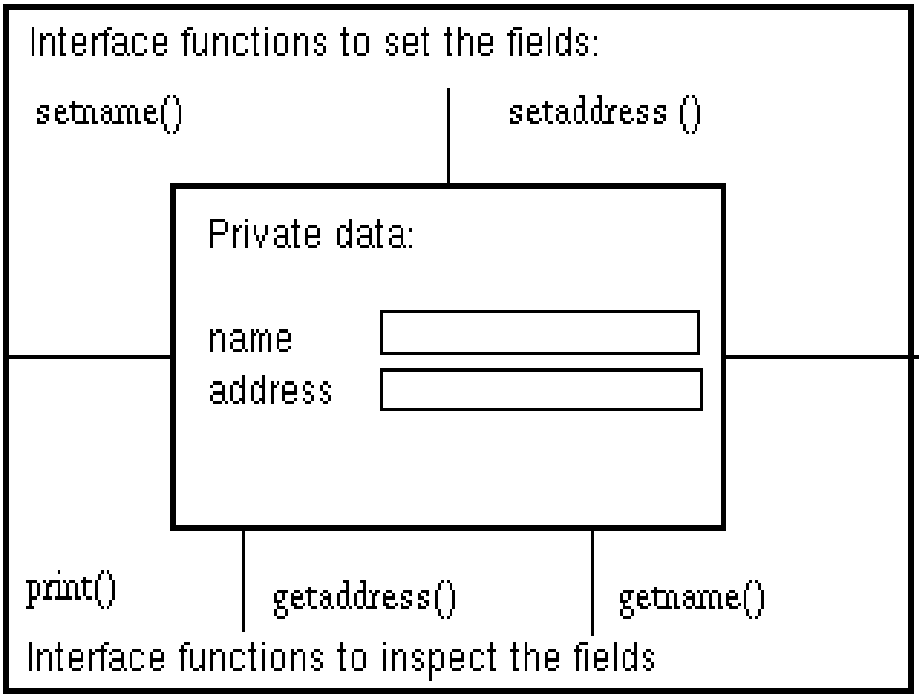
```
void Person::setname (char const *n)
{
  strncpy (name, n, 79);
  name[79] = '\0';
}

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

Strukturen in C und in C++

Datenkapselung: `public`, `private` und `class`

Die Datenkapselung kann hier folgendermaßen illustriert werden:



Klassen

Beispiel für eine Klassendeklaration:

```
class Person
{
  public:       // Zugriffsfunktionen
    void setname(char const *n);
    void setaddress(char const *a);
    void setphone(char const *p);

    char const *getname(void);
    char const *getaddress(void);
    char const *getphone(void);

  private:      // verborgene Elementdaten
    char *name;         // Name der Person
    char *address;      // Adreßfeld
    char *phone;        // Telefonnummer
};
```

Die Klassen*deklaration* wird auch **Schnittstelle** der Klasse (*class interface*) genannt.

Die *Definition* der Elementfunktionen wird auch **Implementierung** der Klasse (*class implementation*) genannt.

Klassen

Konstruktoren und Destruktoren

Der Konstruktor

Der Konstruktor hat per definitionem den gleichen Namen wie die zugehörige Klasse.

Der Konstruktor hat keinen Rückgabetyp, auch nicht `void`.

Deklaration des Konstruktors für die Klasse Person:

```
Person::Person();
```

Der Konstruktor einer Klasse wird vom C++-Laufzeit-system aufgerufen, wenn ein Objekt der Klasse erzeugt wird:

- Für lokale Objekte, die `automatic` Variablen eines Blocks sind, wird der Konstruktor bei **jedem** Eintritt in den Block aufgerufen.

- Für lokale Objekte, die `static` Variablen eines Blocks sind, wird der Konstruktor nur beim **ersten** Eintritt in den Block aufgerufen.

- Für globale Objekte wird der Konstruktor beim Programmstart aufgerufen - **vor** der ersten Anweisung der Funktion `main()`.

Konstruktoren und Destruktoren

Der Konstruktor

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


class Test          // eine Klasse mit Konstruktor
{
  public:           // eine public-Funktion:
    Test();         // der Konstruktor
};

Test::Test()        // Hier ist die Definition.
{
  cout << "constructor of class Test called";
  cout << endl;
}

Test g;             // ein globales Objekt

void func()
{
  Test l;           // ein lokales Objekt in Funktion func()

  cout << "here's function func()" << endl;
}

int main()
{
  Test x;           // ein lokales Objekt in Funktion main()

  cout << "main() function" << endl;
  func();
  return 0;
}
```

Konstruktoren und Destruktoren

Der Konstruktor

In welcher Reihenfolge werden die drei Objekte der Klasse `Test` erzeugt?

- Zuerst wird der Konstruktor für das globale Objekt `g` aufgerufen.

- Danach wird `main()` gestartet, und das Objekt `x` als lokale Variable von `main()` erzeugt.

- Zum Schluss wird `func()` von `main()` aufgerufen, und das Objekt `l` als lokale Variable von `func()` erzeugt.

Das Programm erzeugt also die folgende Ausgabe:

```
constructor of class Test called

constructor of class Test called

constructor of class Test called
```

Konstruktoren und Destruktoren

Der Konstruktor

Anmerkungen zur Definition von Konstruktoren:

- Der Konstruktor hat den gleichen Namen wie die zugehörige Klasse.

- Der Konstruktor hat keinen Rückgabetyp, auch nicht `void`, sowohl in der Deklaration:

```
class Test
{
  public:
    /* hier kein Rückgabetyp */ Test();
};
```

  als auch in der Definition:

```
/* hier kein Rückgabetyp */ Test::Test()
{
  // Anweisungen ...
}
```

- Der Konstruktor im obigen Beispiel hat keine Argumente; er wird daher auch als **Standard-konstruktor** bezeichnet.

- **Allgemeine Konstruktoren** können Argumente haben und wie Funktionen überladen werden.

Konstruktoren und Destruktoren

Der Destruktor

Der Destruktor ist das Gegenstück zum Konstruktor.
Er wird aufgerufen, wenn der Gültigkeitsbereich eines Objekts verlassen wird:

- für lokale nicht-statische Objekte **nach** der letzten Anweisung innerhalb des Blocks, in dem das Objekt definiert ist

- für globale und statische Objekte **nach** dem Verlassen von `main()`

Für die Definition eines Destruktors gelten folgende Regeln:

- Der Destruktor hat den gleichen Namen wie die zugehörige Klasse, aber mit vorangestellter Tilde **~**.

- Der Destruktor hat weder Argumente noch einen Rückgabetyp.

Beispiel:

```
class Test
{
  public:
    Test();       // Konstruktor
    ~Test();      // Destruktor
    // weitere Elementfunktionen
};
```

Klassen

Eine erste Anwendung

Konstruktoren dienen u.a. zur Beschaffung von Speicherplatz, und Destruktoren haben die Aufgabe, diesen Speicherplatz wieder freizugeben.

Für die Klasse Person können wir uns folgendes Szenario vorstellen:

- Der Konstruktor stellt sicher, dass alle Datenelemente mit `NULL` initialisiert werden.

- Der Destruktor gibt sämtlichen allokierten Speicherplatz frei.

- Die Wertzuweisung an ein Datenelement geschieht in zwei Schritten:
    - Zunächst wird evtl. allokierter Speicher freigegeben.
    - Die Zeichenkette, die als Parameter der Funktionen `set...()` übergeben wird, wird dupliziert.

- Die Funktionen `get...()`, die lesend auf die Datenelemente zugreifen, liefern einfach den zugehörigen Zeiger zurück.

Eine erste Anwendung

```
class Person                    // Klassendeklaration
{
  public:
    Person();                   // Konstruktor
    ~Person();                  // Destruktor

    // schreibender Zugriff auf die Datenelemente
    void setname(char const *n);
    void setaddress(char const *a);
    void setphone(char const *p);

    // lesender Zugriff auf die Datenelemente
    char const *getname() const;
    char const *getaddress() const;
    char const *getphone() const;

  private:
    char *name;                 // Name der Person
    char *address;              // Adressfeld
    char *phone;                // Telefonnr.
};

Person::Person()                // Konstruktor
{
  name = NULL;
  address = NULL;
  phone = NULL;
}

Person::~Person()               // Destruktor
{
  delete [] name;
  delete [] address;
  delete [] phone;
}
```

Eine erste Anwendung

```
// schreibender Zugriff: set...()
void Person::setname(char const *n)
{
  delete [] name;
  name = strdupnew(n);
}

void Person::setaddress(char const *a)
{
  delete [] address;
  address = strdupnew(a);
}

void Person::setphone(char const *p)
{
  delete [] phone;
  phone = strdupnew(p);
}

// lesender Zugriff: get...()
char const *Person::getname() const
{
  return name;
}

char const *Person::getaddress() const
{
  return address;
}

char const *Person::getphone() const
{
  return phone;
}
```

Eine erste Anwendung

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

void printperson(Person const &p)
{
  if (p.getname())
    cout << "Name    : " << p.getname() << endl;
  if (p.getaddress())
    cout << "Address : " << p.getaddress() << endl;
  if (p.getphone())
    cout << "Phone   : " << p.getphone() << endl;
}

int main()
{
  Person p;

  p.setname("Linus Torvalds");
  p.setaddress("E-mail: Torvalds@cs.helsinki.fi");
  p.setphone(" - not sure - ");

  printperson(p);
  return 0;
}
```

Klassen

Konstruktoren mit Argumenten

***Allgemeine* Konstruktoren** können Argumente haben.

Beispiel:

```
Person::Person(char const *n,
               char const *a, char const *p)
{
  name = strdupnew(n);
  address = strdupnew(a);
  phone = strdupnew(p);
}
```

Der Konstruktor muss in die Klassendeklaration mit aufgenommen werden:

```
class Person
{
  public:
    Person(char const *n,
           char const *a, char const *p);
    //  ...
};
```

Verwendung des allgemeinen Konstruktors:

```
int main()
{
  Person a("Otto", "Moselstr. 1", "987654"), b;
}
```

Die Reihenfolge der Erzeugung und Zerstörung von Objekten

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

class Test
{
  public:
    Test();
    Test(char const *nm);
    ~Test();
  private:
    char *name;
};

Test::Test() {
  name = strdupnew("without name");
  cout << "Test object without name created\n";
  cout << endl;
}

Test::Test(char const *nm) {
  name = strdupnew(nm);
  cout << "Test object " << name << " created";
  cout << endl;
}

Test::~Test() {
  cout << "Test object " << name << " destroyed";
  cout << endl;
  delete [] name;
}
```

Die Reihenfolge der Erzeugung und Zerstörung von Objekten

```
Test  globaltest("global");

void func()
{
  cout << "start func" << endl;
  Test functest("func");
  cout << "end func" << endl;
}

int main()
{
  Test maintest("main");

  cout << "start main" << endl;
  func();
  cout << "end main" << endl;
  return 0;
}
```

Das Programm erzeugt die folgende Ausgabe :

```
Test object global created

Test object main created

Test object func created

Test object func destroyed

Test object main destroyed

Test object global destroyed
```

Klassen

Die Reihenfolge der Erzeugung und Zerstörung von Objekten

Lokale nicht-statische Objekte werden

- **nach** der letzten Anweisung innerhalb eines Blocks zerstört.

Auch außerhalb von `main()` können einige Aktivitäten stattfinden:

- Falls es globale Objekte gibt, wird ihr Konstruktor **vor** der ersten Anweisung von `main()` aufgerufen.

- Innerhalb des äußersten Blocks von `main()` definierte Objekte werden erst **nach** Verlassen von `main()` freigegeben.

- Wegen der umgekehrten Reihenfolge der Destruktoraufrufe werden globale und statische Objekte zuletzt freigegeben.

Klassen

`const`-Elementfunktionen und `const`-Objekte

Das Schlüsselwort `const` kann verwendet werden, um zuzusichern, dass eine Elementfunktion Datenelemente nicht verändert.

```
class Person
{
  public:
    ...
    // Funktionen zum Lesen der Datenelemente
    char const *getname(void) const;
    char const *getaddress(void) const;
    char const *getphone(void) const;
  private:
    ...
};
```

Die gleiche Spezifikation muss in der Definition der Elementfunktionen angegeben werden.

```
char const *Person::getname() const
{
  // name = strdupnew("Zuweisung nicht erlaubt.");
  return name;
}
```

Klassen

`const`-Elementfunktionen und `const`-Objekte

Auch Objekte können als *konstant* deklariert werden.

Auf konstante Objekte dürfen nur Elementfunktionen angewandt werden, die selbst als konstant vereinbart sind.

Ausnahme: Konstruktoren und Destruktoren

Beispiel:

```
Person const
  karl("Karl", "Rheinstr. 17", "54784");

  // karl.setname("Das ist nicht erlaubt.");
```

Klassen und Dynamische Speicherverwaltung

`new`, `delete` und Zeiger auf Objekte

Beispiele:

```
Person *pp;          // Zeiger auf ein Person-Objekt

pp = new Person;     // hier erzeugt
...
delete pp;           // hier zerstört
```

```
pp = new Person("Hans", "Ostallee 3", "23403");
...
delete pp;
```

```
Person *personarray;

personarray = new Person [10];
...
delete [] personarray;
```

Klassen und Dynamische Speicherverwaltung

Dynamisch allokierte Arrays

Beispiel:

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

class X
{
  public:
    ~X();
};

X::~X()
{
  cout << "X destructor called" << endl;
}

int main()
{
  cout << "start main" << endl;

  X *a = new X[2];
  cout << "Destruction with []'s" << endl;
  delete [] a;

  a = new X[2];
  cout << "Destruction without []'s" << endl;
  delete a;

  cout << "end main" << endl;
}
```

Klassen und Dynamische Speicherverwaltung

Dynamisch allokierte Arrays

Ausgabe des obigen Programms:

```
start main
Destruction with []'s
X destructor called
X destructor called
Destruction without []'s
X destructor called
end main
```

Klassen und Dynamische Speicherverwaltung

Dynamisch allokierte Arrays

Wenn kein Destruktor definiert wird, wird auch keiner aufgerufen.

Beispiel:

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

class X
{
  public:
    ~X();
};

X::~X()
{
  cout << "X destructor called" << endl;
}

int main()
{
  X **a;

  a = new X* [2];

  a[0] = new X [2];
  a[1] = new X [2];

  delete [] a;
}
```

Das Programm erzeugt **keine** Ausgabe!

Klassen und Dynamische Speicherverwaltung

Dynamisch allokierte Arrays

Was tun, wenn die Objekte vom Typ X, auf die die Elemente von a verweisen, auch gelöscht werden sollen?

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

class X
{
  public:
    ~X();
};

X::~X()
{
  cout << "X destructor called" << endl;
}

int main()
{
  X **a;

  a = new X* [2];

  a[0] = new X [2];
  a[1] = new X [2];

  for (int index = 0; index < 2; index++)
    delete [] a[index];

  delete [] a;
}
```

Klassen

`inline`-Elementfunktionen

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

...
Person theo("Theo", "Ruwerstr. 9", "40377");
puts(theo.getname());           // (*)
...
```

Was geschieht in der mit *(*)* gekennzeichneten Zeile?

- `Person::getname()` wird aufgerufen.
- Die Funktion gibt den Wert des Zeigers `name` des Objekts `theo` zurück.
- Dieser Wert, ein Zeiger auf eine Zeichenkette, wird an `puts()` übergeben.
- `puts()` wird aufgerufen und gibt die Zeichenkette aus.

evtl. Performanceproblem:

- **Funktionsaufruf** für den Zugriff auf ein Datenelement

Klassen

`inline`-Elementfunktionen

Deklaration und Definition innerhalb der Klasse

```
// Datei Person.h
class Person
{
  public:
    ...
    char const *getname() const
      { return name; }
    ...
};
```

Deklaration und Definition innerhalb der Header-Datei

```
// Datei Person.h
class Person
{
  public:
    ...
    char const *getname(void) const;
    ...
};

// inline-Implementierung

inline char const *Person::getname() const
  { return name; }
```

Klassen

`inline`-Elementfunktionen

`inline`-Deklaration außerhalb der Header-Datei

```
// Datei Person.h
class Person
{
  public:
    ...
    char const *getname(void) const;
    ...
};

// Datei Person.cpp
inline char const *Person::getname() const
{
   return name;
}
```

Diese Variante ist nicht empfehlenswert!

Begründung:

siehe Abschnitt `inline`-Funktionen in Kapitel 8

Objekte in Objekten: Komposition

Auch Objekte können als Datenelemente in Klassendefinitionen verwendet werden.

```
class Person
{
  public:
    // Konstruktoren und Destruktor
    Person();
    Person(char const *nm, char const *adr,
           char const *ph, int d, int m, int y);
    ~Person();

    // Zugriffsfunktionen
    void setname(char const *n);
    void setaddress(char const *a);
    void setphone(char const *p);
    void setbirthday(int yr, int mnth, int d);

    char const *getname() const;
    char const *getaddress() const;
    char const *getphone() const;
    int getbirthyear() const;
    int getbirthmonth() const;
    int getbirthday() const;

  private:
    // Datenelemente
    char *name, *address, *phone;
    Date birthday;
};
```

Klassen

Objekte in Objekten: Komposition

Die Zugriffsfunktionen der Klasse `Person` verwenden die Zugriffsfunktionen der Klasse `Date`, um das Geburtsdatum zu manipulieren.

```
int Person::getbirthyear() const
{
  return birthday.getyear();
}
```

Im folgenden allgemeinen Konstruktor der Klasse `Person`

```
Person::Person(char const *nm, char const *adr,
               char const *ph, int d, int m, int y)
{
  name = strdupnew(nm);
  address = strdupnew(adr);
  phone = strdupnew(ph);

  birthday.setday(d);
  birthday.setmonth(m);
  birthday.setyear(y);
}
```

- wird zuerst implizit der Standardkonstruktor der Klasse `Date` aufgerufen,
- und anschließend wird `birthday` durch Aufrufe von Elementfunktionen der Klasse `Date` initialisiert.

Das ist ineffizient.

Objekte in Objekten: Komposition

Initialisierungslisten

```
Person::Person(char const *nm,
               char const *adr,
               char const *ph,
               int d, int m, int y)
  : birthday(d, m, y)
{
  name = strdupnew(nm);
  address = strdupnew(adr);
  phone = strdupnew(ph);
}
```

Die Initialisierungsliste wird zwischen der Argumentliste und dem Rumpf des Konstruktors `Person::Person()` angegeben.

Die Initialisierungsliste wird **vor** dem Rumpf des Konstruktors abgearbeitet.

Objekte in Objekten: Komposition

Initialisierungslisten

Initialisierungslisten sind notwendig, wenn eine Klasse Datenelemente enthält, die als `const` deklariert sind.

```
class X {
  public:
    X(int d = 0) { data = d; }    // Fehler!!
  private:
    const int data;
};
```

Die Zuweisung an `const`-Datenelemente ist nicht erlaubt!

Abhilfe:

```
class X {
  public:
    X(int d = 0) : data(d) {}
  private:
    const int data;
};
```

Verwendung des Konstruktors z.B. folgendermaßen:

```
int main() {
  X x;
  X y(10);
  X z = 20;
}
```

Objekte in Objekten: Komposition

Initialisierungslisten

Initialisierungslisten sind notwendig, wenn eine Klasse Datenelemente enthält, die als **Referenz** deklariert sind.

```
class Y {
  public:
    Y(int &r) { ref = r; }        // Fehler!!
  private:
    int &ref;
};
```

Mit der Anweisung `ref = r;` sollte die Referenz `ref` initialisiert werden. Der Compiler interpretiert `ref = r;` aber als Wertzuweisung an die Variable, auf die `ref` verweist, und beschwert sich über die fehlende Initialisierung von `ref`.

Abhilfe:

```
class Y {
  public:
    Y(int &r) : ref(r) {}
  private:
    int &ref;
};
```

Objekte in Objekten: Komposition

Initialisierungslisten

Ein kleines Quiz:

Wodurch unterscheiden sich die folgenden beiden Klassendeklarationen?

```
class Y {
  public:
    Y(int &r) : ref(r) {}
  private:
    int &ref;
};

class Z {
  public:
    Z(int r) : ref(r) {}
  private:
    int &ref;
};
```

`friend`-Funktionen und `friend`-Klassen

`private`-Daten und `private`-Funktionen einer Klasse können normalerweise nur in Code verwendet werden, der Bestandteil der Klasse ist.

```
class A
{
  public:
    A(int v) { value = v; }
    int getval() { return value; }

  private:
    int value;
};

void decrement(A &a)
{
  a.value--;                    // Fehler!!
}

class B
{
  public:
    void touch(A &a)
    {
      a.value++;                // Fehler!!
    }
};
```

`friend`-Funktionen und `friend`-Klassen

```
class A
{
  public:
    // B ist mein Freund, ich vertraue ihm.
    friend class B;

    // decrement() ist auch ein guter Kumpel.
    friend void decrement(A &what);

    ...
};
```

Einige Anmerkungen zur `friend`-Deklaration:

- Die `friend`-Deklaration ist weder symmetrisch noch transitiv.

- Da die `friend`-Deklaration die Datenkapselung durchlöchert, sollte man sparsam damit umgehen.

- Klassen und Funktionen, die als `friend` einer Klasse K deklariert werden, sind abhängig von der Implementierung dieser Klasse K.

- Im Allgemeinen sollten `friend`-Funktionen und `friend`-Klassen nicht verwendet werden.