

C Programmierung in C++

C.1 Überblick

- C++ im Vergleich zu C und Java
- Objekte und Klassen in C++
- Konstruktoren und Destruktoren
- Vererbung
- Ausnahmebehandlung (Exceptions)
- Überladung von Operatoren
- Templates
- Hier nicht betrachtet: Standard Templates Library (STL)

Übungen zu "Verteilte Systeme"

C.1

C.1 Überblick

1 Geschichte von C++

- 1980: Dennis Ritchie erweitert C zu *C mit Klassen*
- 1983: Bjarne Stroustrup führt C++ V1.0 ein
- 1986: Bjarne Stroustrup veröffentlicht *The C++ Programming Language* (1st Edition)
- 1989: ANSI verabschiedet Standard C mit Elementen von C++
- 1989: ANSI-Komitee X3J16 beginnt mit Standardisierung von C++ (V2.0)
- 1991: *The Annotated C++ Reference Manual* definiert C++ V3.0, inklusive *Templates* und *Exceptions*
- 1993: C++ V3.1 führt *Namespaces* und *Run-Time Type Identification* ein
- 1997: ISO WG21 und ANSI X3J16 übernehmen C++ und die *Standard Template Library (STL)* als Standard ISO/IEC FDIS 14882

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2005

CPP.fm 2005-04-27 08:55

C.2

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Was ist C++?

- Eine Obermenge von C
- Ein "besseres" C
 - ◆ Strenge Typprüfung
 - ◆ Prototypen
 - ◆ Überladen von Funktionen und Operatoren
- Erweiterung von C um objektorientierte Konzepte
 - ◆ Objekte
 - ◆ Klassen
 - ◆ Vererbung
 - ◆ Polymorphismus
- *Aber:* C++ erzwingt keine objektorientierten Programmwurf!
(anders als Java...)

Übungen zu "Verteilte Systeme"

C.3

C.1 Überblick

3 Literatur

- Bjarne Stroustrup: „The C++ Programming Language, Special Edition Addison-Wesley, Reading MA, 2000.
- *ANSI C++ Public Comment Draft*, December 1996.
- C++ Programmieren mit Stil, dpunkt-Verlag, 2003
- C++ - Einstieg für Anspruchsvolle, Addison-Wesley, 2005
- Scott Meyers: *Effective C++*, 2nd Edition, Addison-Wesley, Reading MA, 1997.
- Scott Meyers: *More Effective C++*, Addison-Wesley, Reading MA, 1995.

Übungen zu "Verteilte Systeme"
© Universität Erlangen-Nürnberg • Informatik 4, 2005

CPP.fm 2005-04-27 08:55

C.4

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

C.2 Grundlagen

- Ein- und Ausgabe über Ströme
- Inlining
- Scope-Operator
- Namensräume
- Speicherverwaltung
- Überladen von Funktionen
- Referenzvariablen
- Default-Parameter
- Konstanten

VS - Übung

Übungen zu "Verteilte Systeme"

C.5

1 Ein- und Ausgabe

C.2 Grundlagen

- Eingabe und Ausgabe zu Strömen (*Streams*) via *Operatoren*
 - ◆ `cin` Eingabestrom (global)
 - ◆ `cout, cerr, (clog)` Ausgabeströme (global)
 - ◆ `>>` Eingabe-Operator
 - ◆ `<<` Ausgabe-Operator

- Beispiel:

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

void main() {
    int test; // i/o test variable
    cin >> test;
    cout << "test=" << test << endl;
}
```

VS - Übung

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08:55

C.6

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

2 Inlining

- Reserviertes Schlüsselwort `inline`:

```
inline return_type function_name( parameter_list ) {
    function_body
}
```

- ◆ Compiler versucht Funktionsaufruf zu optimieren
- ◆ Anstelle eines Funktionsaufrufs wird die komplette Funktion beim Aufrufer eingefügt → Schnellere Aufrufe, aber größere Programme
- ◆ Weitere Optimierungen möglich (z.B. bei Aufrufen mit konstanten Parametern)
- ◆ Nicht möglich bei rekursiven Funktionen
- ◆ *Implementierung der Funktion muss in der Header-Datei stehen (.H or .hh)!!!*
- Unterschiede zu Präprozessor-Makros (`#define`):
 - ◆ Makros werden als normaler Text expandiert
 - ➔ Keine Typ-Prüfung, oft mysteriöse Syntaxfehler
 - ◆ Makros können rekursiv expandiert werden

VS - Übung

Übungen zu "Verteilte Systeme"

C.7

3 Scope-Operator

C.2 Grundlagen

- Neuer Operator `::`
- Vor allem mit Klassen und Namensräumen benutzt
- *Hier*: Zugriff auf versteckte Variable in anderem Scope
- Beispiel:

```
#include <iostream>

int test = 4711; // globale Variable

void main() {
    int test = 1234; // lokale Variable

    cout << "Die globale Variable ist " << ::test << "\n";
    cout << "Die lokale Variable ist " << test << "\n";
}
```

VS - Übung

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08:55

C.8

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

4 Namensräume

■ Neues Schlüsselwort `namespace`:

```
namespace namespace_name {
    // Deklarationen/Definitionen...
}
```

- ◆ Öffnet einen neuen Namensraum für Bezeichner
- ◆ Namensräume können geschachtelt verwendet werden
- ◆ Zugriff über den Scope-Operator `::`
- ◆ Ähnlich `package` in Java, aber kein Bezug zur Datei-Organisation

■ Beispiel:

```
namespace Date {
    struct Time {
        int year;
        ...
    };
}
```

4 Namensräume (2)

■ Import von Bezeichner anderer Namensräume via `using`:

```
using namespace_name::identifer;
```

- ◆ Wie "`import package.identifer;`" in Java

■ Import von kompletten Namensräumen:

```
using namespace namespace_name;
```

- ◆ Wie "`import package.*;`" in Java

■ Beispiel:

```
namespace Date {
    struct Time { ...
    };
}

namespace MyApp {
    using Date::Time;
    Time today;
}
```

5 Speicherverwaltung

■ Zwei Operatoren in C++:

◆ Speicherreservierung mit `new`

```
type *pointer_to_type;
pointer_to_type = new type;
```

- ▶ Falls Reservierung fehlschlägt, wird eine `std::bad_alloc` Exception geworfen (oder ein `NULL`-Zeiger zurückgegeben)
- ▶ vgl. C: Kein expliziter Cast auf Typ notwendig

◆ Speicherfreigabe mit `delete`

```
delete pointer_to_type;
```

- ▶ Programmierer verantwortlich für die Freigabe
- ▶ Auf Zeiger kann auch nach Freigabe noch zugegriffen werden
- ▶ Häufige Quelle für Programmierfehler
- ▶ `delete` für einen `NULL`-Zeiger ist erlaubt

■ vgl C: Speicherverwaltung mit `malloc` und `free`

5 Speicherverwaltung (2)

■ Beispiel:

```
int *x=0; // okay
delete x; // okay
x = new int; // okay
delete x; // okay
delete x; // falsch
```

■ Spezielle Syntax für Arrays:

```
int *ap = new int[7];
delete[] ap; // nicht: delete ap !!!
```

- *Niemals* `malloc` / `free` mit `new` / `delete` mischen!
 - Achtung: Z.B. `strdup` macht implizit ein `malloc`
- Keine *Garbage Collection* in C++

6 Überladen von Funktionen

- Gleicher Funktionsname für unterschiedliche Implementierungen
 - ◆ Funktioniert für reine C-Funktionen und für C++ Methoden
- Überladene Funktionen werden unterschieden durch:
 - ◆ Anzahl der Parameter
 - ◆ Typ der Parameter
 - ◆ Reihenfolge der Parametertypen
 - ◆ *Nicht*: Rückgabe-Typ (Rückgabe-Wert kann ignoriert werden)

- Beispiel:

```
void Print();           // okay
void Print(int, char*); // okay
int Print(float);      // okay
int Print();           // fehler, nicht unterscheidbar
```

7 Referenzvariablen

- Adress-Operator & in Variablendeklaration


```
type &reference_variable = variable_of_type;
```
- Referenzvariablen
 - ◆ Keine eigenständigen Variablen
 - ◆ Proxy oder Alias für eine andere Variable
 - ◆ Muss bei Deklaration initialisiert werden (mit einem *Ivalue* - etwas, was auf der linken Seite einer Zuweisung sein kann)

- Beispiel:

```
int x = 5;           // Variable
int &rx = x;        // Referenz auf x
x = 6;              // x==6 und rx==6
```

- Operationen auf Referenzvariablen verändern die referenzierte Variable
- Ähnlich zu Zeigern mit impliziter Dereferenzierung

7 Referenzvariablen (2)

- Referenzparameter
 - ◆ Ermöglichen implizite *call-by-reference*-Semantik
 - ◆ Keine Zeiger notwendig
 - ◆ Aufrufer verwendet normale Aufrufsyntax
 - ◆ Nachteil: Syntax des Aufrufs zeigt nicht die Semantik

- Beispiel:

```
#include <iostream>
using std::cout;

void increment(int& x) {
    x++;
}

void main() {
    int x = 5;
    increment( x );
    cout << "x=" << x << "\n"; // x==6
}
```

7 Referenzvariablen (3)

- Rückgabe von Referenzen ebenfalls möglich
- Funktion liefert eine Variable (*Ivalue*), nicht einen Wert

```
int global = 0;           // globale Variable
int& func() {
    return global;       // Rückgabe: Referenz auf global
}

int main() {
    int x;
    x = func() + 1;     // x = global + 1;
    func() = x;         // global = x;
}
```

- Rückgabe von Referenzen auf lokale Variablen verboten

```
int& func() {
    int x = 0;
    int& rx = x;
    return rx;          // nicht erlaubt!
}
```

8 Default-Parameter

- Funktionsparameter können einen *Defaultwert* besitzen
- Wird verwendet, wenn der Parameter im Aufruf fehlt
 - ➔ Nur am Ende der Parameterliste erlaubt
- Beispiel:

```
void print(char* string, int nl = 1);

print( "Test", 0 );
print( "Test" ); // äquivalent zu print( "Test", 1 )
print();        // falsch, Parameter char* fehlt
```

- Achtung: Überladen und Defaultparameter können Mehrdeutigkeit verursachen

```
void print(char* string);
void print(char* string, int nl = 1);
print( "Test" ); // welche Funktion ist gemeint??????????
```

9 Konstanten

- Reserviertes Wort `const` ändert Deklaration
 - ◆ `const`-Variablen sind nur lesbar (`final` in Java)
 - ◆ Initialisierung erfolgt bei Deklaration
- Beispiele:

```
const int k = 42;
char* const s1 = "Test1";
const char* s2 = "Test2";
const char* const s3 = "Test3";

k = 4; // Fehler: k ist const
s1 = "New test"; // Fehler: Zeiger ist const
*s1 = 'P'; // okay, Zeichen von s1 sind nicht const
s2 = "New test"; // okay, Zeiger selbst ist nicht const
*s2 = 'P'; // Fehler: Zeichen von s2 sind const<
```

- Vorzuziehen gegenüber `#define`, weil vom Compiler verwaltet
 - ◆ Definition von lokalen Konstanten; typgebunden
 - ◆ Zeiger auf Konstanten möglich (wie Zeiger auf Variablen)

C.3 Objekte und Klassen in C++

- Erweiterung von `struct`
- Klassen
- Sichtbarkeit
- Erzeugung von Objekten
- Zugriff auf Objekte
- Objekt-Methoden

1 Erweiterung von struct

- Neues Konzept für `struct`
 - ◆ Jede `struct` definiert einen Typ
 - ◆ Lokale Funktionen in `struct`
- Beispiel:

```
struct Person
{
    char*   name;
    int     age;

    void    setName( char* );
    void    setAge( int );
};
```

- Nachteil: uneingeschränkter Zugriff auf alle Teile von aussen

2 Klassen

- Klassendeklaration in C++ mit Schlüsselwort `class`:

```
class Klassenname {
    Deklaration von Variablen und Funktionen
};
```

- ◆ Enthält Deklaration von Daten und Methoden (in C++ *members* genannt)

- Beispiel:

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

    void    setName( char* );
    void    setAge( int );
};
```

3 Sichtbarkeit

- Unterschiedliche Sichtbarkeit für Teile eines Objekts:
 - ◆ `private`: Zugriff nur innerhalb der Klasse
 - ◆ `public`: Zugriff von überall
 - ◆ `protected`: wie `private`, aber abgeleitete Klassen haben Zugriff
- Teile können in beliebiger Reihenfolge deklariert werden (und auch wiederholt werden)
- Der `public`-Teil stellt die Schnittstelle für andere Objekte dar
- Standard-Sichtbarkeit ist `private` !

3 Sichtbarkeit (2)

- Beispiel:

```
class Person {
private:
    char*   name;           // private member variables
    int     age;

public:
    void    setName( char* ); // public member functions
    void    setAge( int );
};
```

4 Objekterzeugung

- Syntax ist identisch zur Deklaration einer Variablen
- *Statische* Erzeugung:


```
Person peter;
Person john;
```

 - ◆ Objekt wird gelöscht, wenn der Scope des Bezeichners verlassen wird
- *Dynamische* Erzeugung:


```
Person* peter;
peter = new Person; // Objekt wird erzeugt
```

 - ◆ Objekt muss explizit gelöscht werden


```
delete peter; // Objekt wird gelöscht
```

5 Zugriff auf Objekte

- Zugriff von ausserhalb des Objekts
 - ◆ Private-Variablen und -Funktionen sind nicht zugänglich
 - ◆ Public-Variablen und -Funktionen sind zugänglich
- Zugriffsoperator
 - ◆ Wie bei `struct` mit dem Punkt-Operator `.`
 - ◆ Bei Zeigern auf Objekte: Pfeil-Operator `->`
- Beispiel:

```
Person peter;
Person* john = new Person;

peter.setName( "Peter Smith" ); // okay, public
cout << peter.name;           // falsch, private
john->setAge( 35 );             // okay, public
cout << john->age;             // falsch, private
delete john;
```

Übungen zu "Verteilte Systeme"

C.25

6 Objektmethoden

- Definition *innerhalb* der Klassen-Deklaration:
 - ◆ Implementierung folgt direkt auf die Deklaration (wie in Java)
 - ◆ Funktion wird automatisch `inline`
 - ◆ Normalerweise in Header-Dateien verwendet (`.h`, `.H` oder `.hh`)
- Definition *ausserhalb* der Klassen-Deklaration:
 - ◆ Innerhalb der Klasse lediglich Deklaration des Funktionsprototypen
 - ◆ Zur Definition muss zunächst der Klassenname genannt werden, gefolgt vom Funktionsname, abgetrennt durch den Scope-Operator `::`
 - ◆ Normalerweise in Implementierungsdateien (`.C`, `.cc`, oder `.cpp`)

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08.55

C.26

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

7 Objektmethoden (2)

- Beispiel:

◆ Header (Person.h)

```
#ifndef PERSON_H
#define PERSON_H
class Person {
private:
    char*   name;
    int     age;
public:
    void    setName( char* n ) {                // inline
        name = n;
    }
    void    setAge( int );
};
#endif
```

◆ Implementation (Person.cpp)

```
#include "Person.h"

void Person::setAge( int i ) {
    age = i;
}
```

Übungen zu "Verteilte Systeme"

C.27

8 Konstante Objekte

- Mit "const" deklarierte Variable
 - ◆ Initialisierung bei Deklaration
 - ◆ Können danach nicht mehr verändert werden
 - ◆ Nützlich für Methodenparameter
- Dummes Beispiel:


```
const Person nobody;
```
- Nur Operationen, die das Objekt nicht verändern, dürfen aufgerufen werden
- Wie weiß der Compiler das?
 - ◆ Er weiß es nicht!
 - ◆ Er braucht einen Hinweis vom Programmierer!

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08.55

C.28

Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

8 Konstante Objekte (2)

- Methoden können mit `const` deklariert werden
- `const`-Methoden verändern das Objekt, an dem sie aufgerufen werden, nicht
- Beispiel:

```
class Person {
private:
    char*   name;
    int     age;

public:
    int getAge() const {
        return age;
    }
};
```

C.4 Konstruktoren und Destruktoren

- Konstruktoren
- Destruktoren
- Member-Objekte
- Kopier-Konstruktor
- Objekt-Arrays

1 Konstruktoren

- Wie in Java
- Definiert als Methode der Klasse
- Name der Methode ist identisch zum Namen der Klasse
- *Kein* Rückgabewert (nicht einmal `void`)
- Mehrere Konstruktoren durch Überladen
- Deklaration normalerweise im `public`-Teil der Klasse
- Zweck: Automatische Initialisierung des neuen Objekts nach Erzeugung
 - Konstruktor muss Objekt in einen konsistenten Zustand bringen
- Compiler erzeugt einen minimalen Default-Konstruktor (ohne Argumente), falls keiner in der Klasse deklariert wurde

1 Konstruktoren (2)

- Aufgerufen bei:
 - ◆ Erzeugung eines Objekts über den Operator `new`
 - ◆ Erzeugung eines statischen Objekts
- Minimaler Default-Konstruktor (vom Compiler erzeugt):

```
Person::Person() {}
```

- Standard-Konstruktor (ersetzt minimalen Konstruktor):

```
Person::Person() {
    name = NULL;
    age = 0;
}
```

1 Konstruktoren (3)

- Andere Konstruktoren:

```
Person::Person( char *n, int i = 0 ) {
    name = n;
    age = i;
}
```

- ◆ Default-Werte sind möglich

2 Destruktoren

- Ähnlich zu `finalize` in Java
- Deklariert als Klassenmethode
- Methodenname ist der Klassenname mit `~` davor
- *Kein* Rückgabetyt (nicht einmal `void`)
- Nur *eine* Destruktor möglich
- Destruktoren haben *keine* Parameter
- Deklaration normalerweise im `public`-Teil der Klasse
- Zweck: Aufräumen vor dem Löschen des Objekts
- Compiler erzeugt Default-Destruktor (tut nichts), falls in der Klasse keiner deklariert

2 Destruktoren (2)

- Aufgerufen bei:
 - ◆ Löschen eines Objekts über den Operator `delete`
 - ◆ Verlassen des Scopes eines statischen Objekts
- Minimaler Default-Destruktor (erzeugt vom Compiler):

```
Person::~Person() {}
```

3 Member-Objekte

- Objekte anderer Klassen als Mitglieder innerhalb einer Klasse
- ```
class Workplace {
 Person worker;
 ...
};
```
- Zugriff ganz normal mit den Operatoren `.` und `->`
  - Probleme bei der Initialisierung:
    - ◆ Werden die Konstruktoren der Member-Objekte aufgerufen?
    - ◆ Wenn ja, wann werden diese aufgerufen?
    - ◆ Welche Konstruktoren werden aufgerufen?
    - ◆ Welche Parameterwerte werden verwendet?
  - Ähnliches Problem beim Löschen von Objekten:
    - ◆ Wann werden die Destruktoren der Member-Objekte aufgerufen?
    - ◆ Kein Problem: Nur ein Destruktor, keine Parameter

### 3 Member-Objekte (2)

- Definition einer Initialisierungsliste im Konstruktor:

```
class_name::class_name(parameter_list)
 : member1(parameters), member2(parameters), ...
{ ...
}
```

- Beispiel:

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

class Workplace {
 Person worker;
 ...
};

Workplace::Workplace(char* name)
 : worker(name)
{ ... }
```

### 4 Kopier-Konstruktor

- Wann wird der Kopier-Konstruktor verwendet?
  - Objekt ist ein Parameter bei einem Funktionsaufruf (als *call-by-value*)
  - Objekt ist ein Rückgabewert einer Funktion
  - Initialisierung eines Objekts mit einem existierenden Objekt

```
Person peter(john);
```

- Beispiel:

```
Person::Person(const Person& p) {
 name = p.name;
 age = p.age;
}
```

- Wichtig: Referenz-Operator & verwenden
- Default-Kopier-Konstruktor (vom Compiler erzeugt) kopiert bit für bit

### 5 Arrays von Objekten

- Statische Arrays

- ◆ Ohne Initialisierung

- ➔ Für alle Elemente wird der Standard-Konstruktor aufgerufen

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

- ◆ Mit Initialisierung

- ➔ Initialisierungsausdrücke werden für die ersten Elemente verwendet, ggf. wird für den Rest der Standard-Konstruktor aufgerufen

```
Person test[4] =
{ "Peter", Person("John") };
// test[0] und test[1]: Person::Person(char*)
// test[2] und test[3]: Person::Person()
```

### 5 Arrays von Objekten (2)

- Dynamisch erzeugte Arrays

- ◆ Der Default-Konstruktor wird immer aufgerufen

```
Person *table;
table = new Person[4]; // 4 times Person::Person()
```

- Zugriff wie gewohnt über den Operator [ ]

```
Person table[4];

table[0].setName("Peter");
```

- Zerstören von Arrays

- ◆ Für alle Elemente wird der Destruktor aufgerufen
- ◆ Dynamisch allokierte Arrays müssen über `delete[]` gelöscht werden

## C.5 Vererbung

- Einfache Vererbung
- Scope-Operator
- Modifikation der Sichtbarkeit
- Konstruktoren und Destruktoren
- Type-Casting
- Virtuelle Methoden
- Polymorphismus
- Virtuelle Destruktoren
- Abstrakte Basisklassen
- Mehrfachvererbung

### 1 Vererbung

- Ähnlich wie in Java
- Zweck: Verwendung existierender Implementierungen (Klassen)
- Neue Klasse *erbt* Eigenschaften einer existierenden Klasse
- Notation:
  - ◆ Klasse, die erbt: Unterklasse (Subclass)
  - ◆ Klasse, von der geerbt wird: Oberklasse oder Basisklasse (Superclass oder Baseclass)
- In C++: *Ableitung* neuer Klassen von existierenden
- Ableitung/Vererbung ist eine "ist-ein"-Beziehung
- Eine Basisklasse: Einfache Vererbung; sonst: Mehrfachvererbung

## 1 Vererbung (2)

- Syntax:

```
class subClass :
[modifier] superClass1, [modifier] superClass2, ... {
 Deklaration von neuen Member-Variablen und
 neuer oder überschriebener Member-Funktionen (Methoden)
```

- Nicht vererbt werden:
  - ◆ Konstruktoren
  - ◆ Destruktor
  - ◆ Zuweisungs-Operator

### 1 Vererbung (3)

- Generell: Alles was nicht überschrieben wird, wird geerbt

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

```
class Employee : public Person { ...
public:
 void print();
 void setSalary(float);
};
```

verhält sich wie

```
Class Employee : public Person { ...
public:
 void print(); // from Employee
 void setName(char*); // from Person
 void setSalary(float); // from Employee
};
```

## 2 Scope-Operator

- Oft wird Zugriff auf überschriebene Methode der Basisklasse benötigt

- Scope-Operator `::`

```
class_name::method(...)
```

- Kein `super` wie in Java!

- Beispiel:

```
class Employee : public Person { ...
public:
 void print() {
 // print(); // Nein! => Endlosschleife!
 Person::print();
 cout << "Salary:" << salary << "\n";
 }
};
...
Employee a;
a.print();
a.Person::print();
```

## 3 Modifikation der Sichtbarkeit

- Spezifikation, wie Elemente der Basisklasse in der abgeleiteten Klasse sichtbar sein sollen

- `public`-Schlüsselwort bei Vererbung:

- ◆ `public` bleibt `public`
- ◆ `protected` bleibt `protected`
- ◆ `private` nicht zugreifbar in abgeleiteter Klasse

- `protected` / `private` -Schlüsselwort bei Vererbung:

- ◆ `public` wird zu `protected` / `private`
- ◆ `protected` wird zu `protected` / `private`
- ◆ `private` nicht zugreifbar in abgeleiteter Klasse

## 3 Modifikation der Sichtbarkeit (2)

- Normalerweise wird nur `public`-Vererbung verwendet

- `protected` und `private` ändern die Schnittstelle

➔ **abgeleitete Klasse ist kein Untertyp der Oberklasse mehr!**

- Default ist `private` !

## 4 Konstruktoren

- Initialisierung der Elemente der Basisklasse durch Konstruktoren der Basisklasse

- Konstruktor der abgeleiteten Klasse ruft Konstruktoren der Basisklassen via *Initialisierungsliste* auf

```
class_name::class_name(parameter_list)
 : superclass1(parameters), superclass2(parameters), ...
```

- Konstruktoren der Basisklassen werden vor dem Konstruktor der abgeleiteten Klasse ausgeführt

- Beispiel:

```
Employee::Employee(char* n, int a, float s)
 : Person(n, a), salary(s)
{
 ...
}
```

## 5 Destruktoren

- Löschen von Elementen der Basisklasse muss im Destruktor der Basisklasse erfolgen
- Destruktor der Basisklasse wird *automatisch* aufgerufen *nach* dem Destruktor der abgeleiteten Klasse (andere Reihenfolge als beim Konstruktor)
- Beispiel:

```
Employee::~Employee()
{
 //Destroy only new members in employee
}
```

## 6 Zeiger auf Objekte

- Zeiger auf ein Objekt einer abgeleiteten Klasse kann einem Zeiger auf eine Basisklasse zugewiesen werden
  - ◆ Unterklasse ist eine Erweiterung der der Basisklasse, und daher auch ein Untertyp
- Anders herum funktioniert es nicht
  - ◆ Explizites *type casting* notwendig
- Allgemeine Regel:
  - Speziellere Typen können einem allgemeineren Typ zugewiesen werden
- Zeiger haben einen *statischen* und einen *dynamischen* Typ:
  - ◆ statisch: Klasse von der Zeigerdeklaration
  - ◆ dynamic: Klasse des Objekts, auf den der Zeiger zeigt
- Statischer Typ legt die zugängliche Schnittstelle fest (Mitglieder und Methoden)

## 7 Type-Casting

- C-Stil:

```
Class Person { ... };
Class Employee : public Person { ... };
...
Employee* e = new Employee; // okay
Person* p = new Person; // okay
Person* pe = e; // okay
Employee* e1 = p; // Compiler-Fehlermeldung
Employee* e2 = pe; // Compiler-Fehlermeldung
Employee* e3 = (Employee*) pe; // okay
Employee* e4 = (Employee*) p; // nicht erkennbarer Fehler
```

- ◆ Compiler überprüft den dynamischen Typ nicht
- ◆ Vor ANSI-C++ gab es keine Typinformation zur Laufzeit (Run-Time Type Information, RTTI)
- ◆ Vermeiden !!!

## 7 Type-Casting (2)

- In ANSI-C++ gibt es vier neue Methoden für Type-Casts:

```
type variable = static_cast<type>(parameter);
type variable = reinterpret_cast<type>(parameter);
type variable = dynamic_cast<type>(parameter);
type variable = const_cast<type>(parameter);
```

- **reinterpret\_cast:**
  - Zeigertyp in einen beliebigen anderen Zeigertyp umwandeln
  - Der Wert des Zeigers wird dabei nicht verändert
  - Keinerlei Überprüfungen
- **static\_cast:**
  - Nur alle implizit möglichen Casts und deren Umkehrungen unterstützt
  - Bsp: int <-> char, Zeiger auf Basisklasse in Zeiger auf abgel. Klasse
  - Keine echte Überprüfung auf Typkompatibilität

## 7 Type-Casting (3)

- **dynamic\_cast:**
  - ◆ Verwendet Run-Time Type Information, um zu überprüfen, ob der Cast möglich ist
  - ◆ Entspricht dem Casting in Java
  - ◆ Liefert NULL zurück falls nicht möglich, keine Exception geworfen !!!

- Beispiel:

```
class Person { ... };
class Employee : public Person { ... };
...
Employee* e = new Employee;
Person* p = new Person;
Person* pe = e;
Employee* e3 = dynamic_cast<Employee*>(pe); // okay
Employee* e4 = dynamic_cast<Employee*>(p); // liefert NULL
```

## 7 Type-Casting (4)

- **const\_cast:**
  - Änderung der const-Eigenschaft (nur hiermit möglich)

- Beispiel:

```
class C {};
const C * a = new C;
C * b = const_cast<C*>(a);
```

- Kann in folgender Situation nützlich sein:

```
void Print(int *a) {
 cout << *a << endl;
}

int main() {
 const int num = 42;
 Print(const_cast<int *>(&num));
}
```

## 8 Virtuelle Methoden

- Bis jetzt:
  - ◆ Interface-Semantik wird durch Typ eines Zeigers festgelegt (statischer Typ), nicht durch Typ eines Objekts, auf das der Zeiger zeigt (dynamischer Typ)
  - ◆ Zugriff auf alle Elemente einer abgeleiteten Klasse nur nach einem Casting des Zeigers

- Ziel ist *polymorphism*: Ausführung einer geeigneten Methode der Unterklasse, ohne diese explizit zu kennen (*Bei Java immer so!*)

- Lösung: *Virtuelle Methoden*

- ➔ Objekt definiert die Semantik, nicht der Zeiger

- Syntax mit Schlüsselwort **virtual** in der Basisklasse:

```
class class_name {
 virtual return_type method_name(parameter_list)
 { ... }
};
```

## 9 Polymorphism

- Beispiel:

```
class Person { ...
public:
 virtual void print();
};
class Employee : public Person { ...
public:
 void print();
};
...
Person* p = new Person;
Person* pe = new Employee;
p->print(); // Person::print()
pe->print(); // Employee::print()
```

- Aufgerufene Methode wird zur Laufzeit bestimmt
- Aufgerufenes Objekt hat einen definierten Typ, daher ist die aufzurufende Methode eindeutig
- Compiler erzeugt *vtables* (Sprungtabellen für virtuelle Methoden)
  - ◆ Jedes Objekt enthält Zeiger auf vtable der Klasse; Speicherverbrauch!

## 10 Virtuelle Destruktoren

- Dynamisch allokierte Objekte können einem Zeiger der Oberklasse zugewiesen werden
- Problem: Falls Objekt gelöscht wird, wird nur der Destruktor der Oberklasse aufgerufen, wegen des statischen Types des Zeigers
  - ➔ Objekte werden nicht richtig gelöscht

- Lösung: *Virtueller* Destruktor:

```
class class_name {
 virtual class_name::~~class_name()
 { ... }
};
```

- **virtual** muss in der Basisklasse angegeben werden
- Wird bei allen Unterklassen geerbt, auch wenn die Namen der Destruktoren unterschiedlich sind

## 11 Abstrakte Klassen

- Abstrakte Klassen:
  - ◆ Nicht alle deklarierten Methoden werden auch implementiert
  - ◆ Es kann keine Instanzen/Objekte dieser Klasse geben
  - ◆ Unterklassen können nur dann Instanzen haben, wenn alle deklarierten Methoden implementiert werden
- Abstrakte Klassen können verwendet werden
  - ◆ Als Basisklassen ohne Instanzen (**class** mit **abstract**-Methoden in Java)
  - ◆ Zur Definition eines Typs/einer Schnittstelle (**interface** in Java)

- Syntax für nicht implementierte Methoden (*rein virtuell*):

```
class class_name {
 virtual return_type method_name(parameter_list) = 0;
};
```

- Zeiger auf abstrakte Klassen möglich, müssen aber mit Objekten einer nicht abstrakten Unterklasse initialisiert werden

## 12 Mehrfache Vererbung

- Unterklasse hat *mehrfache* Basisklassen (in Java nicht erlaubt)
- Unterklasse enthält implizit *jede* Basisklasse
- Konstruktor der abgeleiteten Klassen kann Konstruktoren von allen Basisklassen in der Initialisierungsliste aufrufen

```
class Base1 { ...
public:
 Base1(int, char*);
};

class Base2 { ...
public:
 Base2(int, float);
};

class Derived : public Base1, public Base2 { ...
public:
 Derived(char *s, int i) :
 Base1(i, s), Base2(i, 4.2) { }
};
```

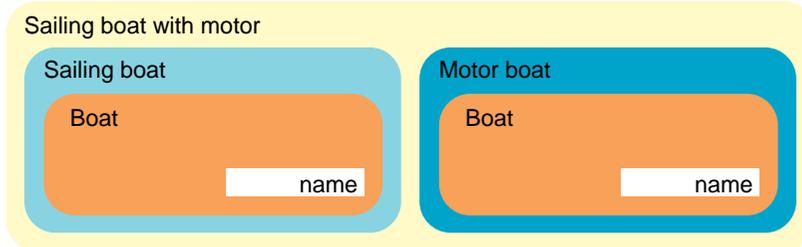
- Wenn ein Objekt einer abgeleiteten Klasse vernichtet wird, werden alle Destruktoren der Basisklassen aufgerufen

## 12 Mehrfache Vererbung (2)

- Problem: *Mehrdeutigkeiten* durch Namenskollisionen
- Zwei oder mehr Basisklassen haben gleiches Element:
  - ◆ Member-Variable mit gleichen Namen
  - ◆ Methoden mit gleichen Namen und gleichen Parametern
- Zuerst automatische Auflösung von Mehrdeutigkeiten, dann Zugriffskontrolle (Sichtbarkeit)
  - ➔ Es nützt nichts, ein Element als **private** zu deklarieren!
- Explizite Auflösung von Namenskollisionen bei Variablen
  - ◆ Angabe des Namens der Basisklasse vor dem Variablennamen mit dem Scope-Operator **::**
- Mögliche Lösung für Methoden:
  - ◆ Methode überschreiben und die gewünschte Methode einer Basisklasse mit dem Scope-Operator auswählen **::** auswählen

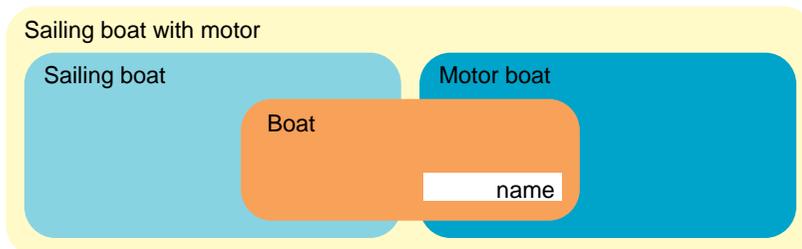
## 12 Mehrfache Vererbung (3)

- Basisklasse enthält Gemeinsamkeiten aller Unterklassen (Generalisierung)
- Problem mit Mehrfachvererbung: Gemeinsame Basisklasse wird mehrfach eingebunden
- Beispiel:



## 12 Mehrfache Vererbung (4)

- Lösung: Implementierung mit einer *virtuellen* Basisklasse
- Beispiel:



- Syntax für *virtuelle Vererbung*:

```
class subClass : virtual public superClasssh {
 Declaration of member variables und functions
};
```

## 12 Mehrfache Vererbung (5)

- Beispiel:

```
Class Boat {
protected: char* name;
public: Boat(char* n) : name(n) { }
};

Class SailingBoat : virtual public Boat {
protected: Sail mySail;
public: SailingBoat(char* n) : Boat(n) { }
};

Class MotorBoat : virtual public Boat {
protected: Motor myMotor;
public: MotorBoat(char* n) : Boat(n) { }
};

Class SailingBoatWithMotor
: public SailingBoat, public MotorBoat {
public: SailingBoatWithMotor(char* n)
: Boat(n), SailingBoat(n), MotorBoat(n)
{ }
};
```

## C.6 Exceptions

- Syntax
- Funktionsweise von Exceptions
- Beispiel: Ressourcen-Allocation
- Unterschiede zu Java
- Exceptions in ANSI C++

## 1 Syntax

- 3 Schlüsselwörter:
  - ◆ `try` versucht den folgenden Block auszuführen
  - ◆ `throw` erzeugt eine Exception und startet die Exception-Behandlung
  - ◆ `catch` fängt eine Exception eines `try`-Blocks auf und verarbeitet den folgenden Block
- Beispiel:

```
try {
 computation
 if error: throw exception_class(...);
}
catch(exception_class variable) {
 exception processing
}
```

## 2 Funktionsweise von Exceptions

- Lineare Verarbeitung der `catch`-Liste
- Gruppierung von Fehlertypen durch Vererbung
  - ◆ Auffangen einer Basisklasse fängt auch alle Unterklassen
- Exceptions werden nach oben weitergereicht, bis ein `catch`-Ausdruck mit passendem Typ gefunden wird
- Alle Destruktoren werden aufgerufen, wenn ein Block über eine Exception verlassen wird
- Falls es keinen passenden `catch`-Ausdruck gibt ➔ Programm wird abgerochen
- `catch( ... )` fängt alle Exceptions

## 3 Unterschiede zu Java

- Kein `finally`
- Exceptions sind *nicht* Teil der Methodensignatur
  - ➔ Können überall geworfen werden
  - ➔ Compiler kann nicht überprüfen, ob alle erzeugten Exceptions irgendwo aufgefangen werden

## 4 Exceptions in ANSI C++

- Funktionen und Methoden *können* eine Exception-Liste angeben
- Schlüsselwort `throw` im Funktionsprototyp:
 

```
return_type method_name (parameter_list) throw (exception_list) {
 Body of method
```
- Ähnlich zu `throws` in Java
- Exception-Liste ist eine Garantie für den Aufrufer
- `std::unexpected( )` wird aufgerufen, falls eine Exception aus der Funktion herausgereicht wird, die nicht in der Liste enthalten ist
- Funktionen ohne Exception-Liste können immer noch beliebige Exceptions erzeugen

## C.7 Weitere Besonderheiten

- Der `this`-Zeiger
- Statische Member-Variablen

### 1 Der `this`-Zeiger

- `this` zeigt auf das aufgerufene Objekt selbst
- Impliziter Parameter bei jedem Methodenaufruf
- Typ: `class_name * const this`
- Falls Methode `const` ist: `const class_name * const this`
- Beispiel:

```
class Person { ...
 char* name;
public:
 void print() { cout << this->name; // = name
 void insertInto(List* l) { l->insert(this) }
 void prettyPrint() {
 cout << "Data: ";
 this->print(); // = print()
 }
};
```

## 2 Statische Member-Variablen

- Normalerweise hat jedes Objekt seine eigene Menge an Variablen
- Ausnahme: Member-Variablen, die als `static` deklariert sind
- Elemente, die als `static` deklariert werden, existieren genau einmal für jede Klasse, egal wie viele Objekte dieser Klasse es gibt
- Ermöglicht es, eine gemeinsame Variable für alle Instanzen einer Klasse zu verwenden
  - ➔ Klassenvariable
- Zugriffsrechte können wie bei Instanzvariablen festgelegt werden

### 2 Statische Member-Variablen (2)

- Globale Initialisierung außerhalb der Klasse
- Beispiel:

```
Class BankAccount {
 static float interestRate;
 ...
};
...
float BankAccount::interestRate = 0.5;
```

- Methoden, die nur auf `static`-Elemente zugreifen, können ebenfalls als `static` deklariert werden
- `static`-Methoden können ohne Objekt aufgerufen werden
- *Kein* `this`-Zeiger, *kein* Zugriff auf Instanzvariablen/-methoden der Klasse

## C.8 Operatoren

- Überladen von Operatoren
- Globale Operatoren
- Operatoren als Members
- Binäre Operatoren
- Unäre Operatoren
- Allokations-Operatoren

### 1 Überladen von Operator

- In C++ (anders als in Java) können Operatoren mit neuen Typen überladen werden
- Funktioniert wie das Überladen von Methoden
- Neues Schlüsselwort **operator**

```
return_type operator operator (parameter_list)
{ ... };
```

- Operatoren, die überladen werden können

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

- Operatoren, die nicht überladen werden können

```
. .* :: ?:
```

- Vorrang und Assoziativität von Operatoren lässt sich nicht ändern

## 2 Globale Operatoren

- Funktionieren wie (globale) Funktionen
- Haben immer das Objekt selbst als Parameter
- Können als "friend" von anderen Klassen deklariert werden => Zugriff auch auf private Variablen
- Beispiel:

```
class Person { ...
 char* name;
 friend ostream& operator << (ostream&, Person);
};

ostream& operator << (ostream& os, Person& p) {
 os << p.name;
 return os;
}
...
Person p("Peter");
cout << p; // call as operator
operator << (cout, p); // call as function
```

### 3 Operatoren als Members

- Operator werden wie eine Klassenmethode behandelt; Zugriff auf alle Members, es gibt einen **this**-Zeiger
- Ein Parameter weniger als der gleiche globale Operator (Objekt via **this**)
- Beispiel:

```
class Complex {
 double real, imag;
public: Complex(double r=0, double i=0)
 : real(r), imag(i) { }
 Complex operator + (const Complex&) const;
};

Complex Complex::operator + (const Complex& c) const {
 Complex result(real+c.real, imag+c.image);
 return result;
}
...
Complex c1, c2, c3;
c1 = c2 + c3; // normal call
c1 = c2.operator + (c3); // generated by the compiler
```

## 4 Binäre Operatoren

- Als globaler Operator: Zwei Parameter
- Als Member: Ein Parameter
- Beispiele (Nur Member-Operatoren):
  - ◆ Zugriffs-Operator
 

```
Class& Class::operator = (Class&)
```
  - ◆ Index-Operator
 

```
element_type& Class::operator [] (index_type)
```

    - Index-Typ normalerweise `int`
  - ◆ Arithmetische Operatoren sowie deren Kombination mit Zuweisung

## 5 Unäre Operatoren

- Als globaler Operator: Ein Parameter
- Als Member: Keine Parameter
- Ausnahme: Postfix-Operatoren
- Beispiele (nur Member-Operatoren):
  - ◆ Prefix increment operator
 

```
class& class::operator ++ ()
```
  - ◆ Postfix increment operator
 

```
class& class::operator ++ (int)
```

    - `int` nur eine Dummy-Parameter zur Unterscheidung von der Prefix-Version
  - ◆ Cast operator
 

```
class::operator target_type ()
```

    - Zieltyp ist zugleich Operatorname und Rückgabetyt

## 6 Allokations-Operatoren

- Eigene Strategien zur Speicherallokation
- Globale Operatoren für alle Klassen
- Operatoren für Allokation auf einer pro-Klassen-Basis
  - ◆ Vorrang vor globalen Operatoren
  - ◆ z.B. Speicherpool für kurzlebige Objekte
- Syntax
  - ◆ Allokations-Operator
 

```
void* operator new (size_t)
```
  - ◆ Deallokations-Operator
 

```
void operator delete (void *)
```

    - ◆ Für Arrays: Operatoren `new[ ]` und `delete[ ]`

## C.9 Templates

- Funktionstemplates
- Klassentemplates

## 1 Funktionstemplates

- Erzeugung generischer Funktionen, die beliebigen Datentyp als Parameter oder Rückgabewert haben können
- Ähnlichkeit zu Makro
- Syntax (kein Unterschied):

```
template <class class_name> function_declaration
template <type type_name> function_declaration
```

- Beispiel

```
template <class GenericType>
GenericTyp max(GenericType a, GenericType b) {
 GenericType result;
 result = a>b?a:b;
 return result;
}

int a=3,b=5,c;
c = max<int>(a,b);
```

Übungen zu "Verteilte Systeme"

C.81

## 1 Funktionstemplates (2)

- Compiler kann die Spezialisierung auch selbst bestimmen

```
c = max(a,b) // Funktioniert im vorhergegangenen Beispiel!
```

- Auch mehrere Parameter möglich

```
template <class T, class U>
T max(T a, U b) {
 return (a<b)?a:b;
}

int i,j;
long l;
i = max(j,l);
```

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08.55

C.82

## 2 Klassentemplates

- Erzeugung generischer Klassen, die durch einen Typ parametrisiert sind
- Beispiel

```
template <class T>
class pair {
 T value1, value2;
public:
 pair (T first, T second) //Konstruktor
 {value1=first; value2=second;}
 T getmax ();
};

template <class T>
T pair<T>::getmax ()
{
 T retval;
 retval = value1>value2? value1 : value2;
 return retval;
}
```

Übungen zu "Verteilte Systeme"

C.83

Übungen zu "Verteilte Systeme"

## 2 Klassentemplates (2)

- Spezialisierung von Templates: einzelne Ausprägungen eines Templates können auch explizit erstellt werden

- Beispiel

```
template <class T>
class pair {...};

template <class T*> // Spezialisierung
class pair<T*>; // es können nur Zeiger verwendet werden

template <>
class pair <int>; // es können nur Integer verwendet werden

template <>
class pair <int>{
 ... // neue implementierung
};
```

- Bei der Deklaration von Objekten und der Auflösung der Überladungen werden spezialisierte Versionen bevorzugt.

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08.55

C.84

Übungen zu "Verteilte Systeme"

CPP.fm 2005-04-27 08.55

## 2 Klassentemplates (3)

- Neben Typen können auch Konstanten als Parameter übergeben werden (z.B. zur Initialisierung)

```
template <class T, int N>
class array {
 T memblock [N];
public:
 setmember (int x, T value);
 T getmember (int x);
};

template <class T, int N>
array<T,N>::setmember (int x, T value) {
 memblock[x]=value;
}

template <class T, int N>
T array<T,N>::getmember (int x) {
 return memblock[x];
}
```

## 2 Klassentemplates (4)

- Verwendung:

```
int main () {
 array <int,5> myints;
 array <float,5> myfloats;
 myints.setmember (0,100);
 myfloats.setmember (3,3.1416);
 cout << myints.getmember(0) << '\n';
 cout << myfloats.getmember(3) << '\n';
 return 0;
}
```

## 3 Template Beispiel

- Fakultäts-Templates

```
template <int X>
struct Faculty {
 static const int result = X * Faculty<X-1>::result;
};

template <>
struct Faculty<1> {
 static const int result = 1;
};

int fak = Faculty<10>::result;
```

## C.10 Zusammenfassung

- Grundlagen
  - ◆ Speicherverwaltung, Überladen von Funktionen, Referenzvariablen
- Objekte und Klassen
  - ◆ Objekterzeugung, konstante Objekte, Konstruktoren, Destruktoren
- Vererbung
  - ◆ virtuelle Methoden, Polymorphism, mehrfach Vererbung
- Ausnahmebehandlung
  - ◆ Unterschiede zu Java
- Operatoren
  - ◆ Überladen, globale vs. Operatoren als Members, Allokations-Operatoren
- Templates
  - ◆ Funktions-, und Klassentemplates