

# **Objekt-Orientierte Programmiersprachen**

Martin Gasbichler, Holger Gast

15. Dezember 2005

---

# C++ – Der Plan

- 15.12. Klassen als benutzer-definierte Datentypen
- 20.12. Objekt-orientierte Programmierung
- 22.12. (Templates)
- Grundlage: Bjarne Stroustrup: “The C++ Programming Language”, (3. Auflage, 1999)

---

# What is C++?

§2.1 C++ is a general-purpose programming language with a bias towards systems programming that

- is a better C
  - supports data abstraction
  - supports object-oriented programming
  - supports generic programming
- 
- C++ ist keine rein objekt-orientierte Sprache
  - C++ will C erweitern um Abstraktionsmechanismen
  - Historisch:
    - Stroustrup kannte Simula 67, musste aber C programmieren
    - Aus C wurde "C with classes", daraus dann C++

---

## Intention von Klassen in C++

§10.3 Small, heavily-used abstractions are common in many applications. Examples are Latin characters, Chinese characters, [. . .] Consequently, mechanisms must be provided for the user to define small concrete types. [. . .] It was an explicit aim of C++ to support the definition and efficient use of such user-defined data types very well.

§10.3.4 I call simple user-defined types [. . .] concrete types to distinguish them from [. . .] class hierarchies and also to emphasize their similarity to built-in types such as int and char. They have also been called value types, and their use value-oriented programming. Their model of use and the “philosophy” begin their design are quite different from what is often advertised as object-oriented programming.

---

## Intention von Klassen in C++

- Klassen definieren neue Typen
- Objekte sollen sich wie Werte eingebauter Typen verhalten
- Das erste Ziel von Klassen ist Effizienz
- Objekte sollen nicht ineffizienter sein als eingebaute Typen
- C++ Klassen unterscheiden sich sehr von Java Klassen
  - Kontrolle über Details Erzeugung und Zerstörung von Objekten
  - Kontrolle über Änderbarkeit von Daten
  - Das Zeiger-Konzept wurde aus C komplett übernommen
  - Keine automatische Speicherverwaltung
  - Kein automatisches Nachladen (weder Compiler noch Laufzeit)

---

# C als Teilmenge von C++

- C++ übernimmt von C
  - Primitive Typen (Eingebaute Typen)
  - Funktionen
  - Anweisungen
  - Ausdrücke
- Neu sind
  - `bool` Typ mit Literalen `true` und `false`
  - Referenztyp (Variation des Zeigertyps)
  - `struct` führt neuen Typ ein
  - `class` Typen und verwandte Konzepte
  - Deklaration von lokalen Variablen überall in Blöcken
  - Namespaces

---

## Einschub: Namespaces

- C hat nur einen globalen Namensraum für Variablen, Funktionen, Typen
- ⇒ Namenskonflikte häufig, Namenskonventionen werden benötigt
- C++ führt **Namensräume** ein

```
namespace a {  
    void print(); // Deklaration  
}  
void a::print() { // Definition  
    ...  
}
```

- Zugriff über **qualifizierten Namen** `a::print` bzw. `b::print`
- `using N::x;` macht `x` aus Namensraum `N` als `x` zugänglich  
“`::`” heißt auch **scope operator**
- `using namespace N;` macht *alle* Bezeichner aus `N` verfügbar  
Gilt als schlechter Stil

---

# Hello World

- Das “Hello-World” Programm sieht in C++ so aus

```
#include <iostream>
using std::cout;
using std::endl;
int main(int argc, char **argv) {
    cout<<"Hello, world!"<<endl;
}
```

- Alternativ:

```
std::cout<<"Hello, world!"<<std::endl;
```

- cout ist ein Ausgabestrom
- Der Aufruf `cout<<x` “schiebt”  $x$  in den Strom

---

## Referenzen

```
int f(int &i) {  
    int &j = i;  
    int k = 41;  
    int &l = k;  
    j = j + 1;  
    l++;  
    cout<<"k = "<<k<<endl; // Ausgabe: 42  
}  
int g() {  
    int n = 9;  
    f(n);  
    cout<<"n = "<<n<<endl; // Ausgabe: 10  
}
```

- Referenzen sind Zeiger: Zugriff auf andere Variable
- Automatische Adresse (Operator &) und Dereferenz (Operator \*)
- Bei Zuweisung: Zuweisung an referenzierten Wert
- Initialisierung mit Referenzen → Alias

---

# Übersetzung von C++ Programmen

---

service.h

---

```
#ifndef SERVICE_H
#define SERVICE_H
void do_something_for_me(int);
#endif
```

service.cpp

---

```
#include "service.h"
#include <iostream>
void do_something_for_me(int i) {
    std::cout<<"Here's my service message: "<<i<<std::endl;
}
```

user.cpp

---

```
#include "service.h"
int main(int argc, char **argv) {
    do_something_for_me(42);
}
```

```
> g++ -c service.cpp
> g++ -c user.cpp
> g++ -o user service.o user.o
```

---

## Charakteristika eingebauter Typen

- Klassen sollen sich wie eingebaute Typen verhalten
- Charakteristika eingebauter Typen
  - Implementierung (Repräsentation) ist verborgen
  - Wertsemantik: Werte werden immer kopiert
  - Compiler kennt Grundoperationen

```
int f(int i) {  
    int j = 2 * i;  
    j = 2 + j;  
    return j + i;  
}  
int g() {  
    return f(42);  
}
```

- Kopie
- Initialisierung
- Zuweisung
- Anlegen von Variablen
- Freigeben von Variablen

- Ziel heute: Übertragung dieser Eigenschaften auf Klassen

---

# Klassen in C++

- `class C { R }` führt neuen Typ `C` ein
- Im Rumpf `R` können stehen
  - Funktionsdeklarationen & -Definitionen (*member functions*)
  - Variablendefinitionen (*member variables*)
  - Typdefinitionen (mit `typedef`)
- Verbergen der Implementierung im `private`: Abschnitt
- Schnittstelle des neuen Typs in `public`: Abschnitt
- `struct` ist Abkürzung für `class ... { public: ... }`
- Member functions haben impliziten `this`-Zeiger

---

## Beispiel: Die `str`-Klasse

- C-Strings mit `char*` sind sehr zerbrechlich
- Wir wollen “Strings als Werte”
- Erster Entwurf:

```
class str {  
    private:  
        char *repr;  
    public:  
        char *get();  
        void set(char *);  
        void set(str);  
        void set_char(int i, char c);  
        char get_char(int i);  
};
```

---

## Einschub: Speicherverwaltung in C++

- `new T` reserviert neuen Speicher für einen  $T$ -Wert
- `delete p` gibt Speicher frei, auf den Zeiger  $p$  zeigt
- `new T[n]` reserviert Speicher für Array mit  $n$  Werten
- `delete [] p` gibt Array frei, auf das Zeiger  $p$  zeigt
- Bemerkungen
  - Anders als in Java muss man Speicher wieder freigeben
  - Niemals `delete []` auf Speicher anwenden, der mit `new` statt `new []` angefordert wurde
  - Freigegebenen Speicher nicht mehr verwenden
  - Speicher nie doppelt freigeben

---

## Die set Funktion von str

```
void str::set(char *s) {
    if (repr) delete [] repr;
    repr = new char[strlen(s)+1];
    strcpy(repr,s);
}
void str::set(str s) {
    set(s.repr);
}
```

- Alten Speicher freigeben, wenn repr nicht der 0-Zeiger ist
- Neuen Speicher für String + abschliessende 0 anlegen
- Inhalt des Strings kopieren
- Für zweite set-Funktion: Einfach erste aufrufen

---

## Probleme bei str

- Initialisierung repr=0 fehlt, sonst macht str::set keinen Sinn!

```
void str::set(char *s) {  
    if (repr) delete [] repr;  
    ...  
}
```

- Brauchen Initialisierungsfunktion  
Anders als Java: Keine Default-Werte für “plain old data types” in C++

```
void str::init() {  
    repr = 0;  
}
```

- Funktion get macht Repräsentation sichtbar

```
char *str::get() { return repr; }
```

---

## Einschub: const-Variablen

- In C und C++ können Werte als const deklariert werden
- Das löst das get() Problem

```
class str {  
    ...  
    public:  
        const char *get(); // Zeiger auf nicht änderbaren String  
    ...  
};  
const char *str::get() { return repr; }
```

- Schreibender Zugriff ist verboten

```
int fail_modify(str s) {  
    s.get()[0] = 'a';  
}
```

```
> g++ -c str.cpp  
str.cpp: In function 'int fail_modify()':  
str.cpp:30: error: assignment of read-only location
```

---

## Problem: Aliasing

- Programmierer muss für jede str-Variable `init` aufrufen
- Funktionsaufruf *call-by-value* funktioniert nicht

```
void break_it(str s1) {
    s1.set("Say goodbye to this world!"); // ändert lokale Kopie
}
void still_ok() {
    str s; s.init();
    s.set("Still ok");
    cout<<"At 1: "<<s.get()<<endl; // liefert "Still ok"
    break_it(s); // soll s nicht ändern
    cout<<"At 2: "<<s.get()<<endl; // soll "Still ok" liefern
    s.set("to the world"); // doppelte Freigabe
}
```

- Problem
  - `s1.repr` wird einfach auf `s.repr` gesetzt → Alias
  - `s1.set()` gibt Speicher frei, auf den `s.repr` noch zeigt

---

## Lösung: Haken und Ösen im Compiler

- Wir müssen dem Compiler mitteilen, wie man `str`-Werte richtig
  - anlegt, damit wir `init` nicht selbst aufrufen müssen
  - an Funktionen übergibt, damit kein Aliasing auftritt
  - freigibt, damit die lokale Kopie `s1` ihren Speicher nach Ablauf von `break_it` wieder freigibt
- Der C++ Ansatz: Den Lebenszyklus von Werten festlegen
  1. Anlegen und initialisieren
  2. Kopieren
  3. Zuweisungen (Überschreiben)
  4. Zerstören
- Idee: Der Programmierer kann für neue Datentypen Funktionen definieren, die der Compiler automatisch an den passenden Stellen im Lebenszyklus aufruft.

---

## Einschub: Überladung

- Überladung: Mehrere Funktionen mit
    - Gleichem Namen
    - Verschiedenen Signaturen
  - **Überladungsauflösung** (*overload resolution*) bei Aufruf für Typen der Argumente
  - Stroustrup: Überlade Funktionsnamen bei logisch äquivalenter Leistung
  - Ebenfalls in Java vorhanden für Methoden
- ⇒ Lebenszyklus verwalten durch Überladung
- Überlade spezielle “Funktionsnamen” für neue Typen
  - Compiler sucht nach passender Funktion bei Erzeugung, Kopie, Zuweisung, . . .

---

## Anlegen und Initialisieren

- Prinzip: Sobald eine Variable angelegt wird, wird sie initialisiert (Einschränkung: Das gilt nicht für “plain old” C-Datentypen)
- Für die Initialisierung eines Datentyps ist ein Konstruktor zuständig.
- Er wird mit `this`-Zeiger auf uninitialisierten Speicher aufgerufen
- Der Konstruktor ohne Argumente heisst **default constructor**  
In Java: *default constructor* ist der vom Compiler erzeugte Konstruktor ohne Parameter
- Er trägt (wie in Java) den Namen der Klasse und hat keine Rückgabe

```
class str {  
    public:  
        str();  
        ...  
};  
void str::str() {  
    repr = 0;  
}
```

---

## Compiler-generierter Default-Konstruktor

- Wenn eine Klasse keine Konstruktoren hat, legt der Compiler einen Default-Konstruktor an
- Dieser ruft die Default-Konstruktoren aller member variables auf, die selbst Klassentypen sind
- Also: “plain old data types” werden *nicht* initialisiert

⇒ Siehe `str::init()` Funktion

---

# Kopieren

- Kopien finden statt bei
    - Funktionsaufruf
    - Rückgabe eines Wertes mit `return`
    - Initialisierung von Variablen (wie `int i = 42;`)
    - Anlegen von temporären Werten
    - (Exception-Verwaltung)
  - Einsicht: Auch in diesen Fällen ist der Speicherplatz für das Ziel der Kopie noch nicht initialisiert!
- ⇒ Der Compiler ruft einen speziellen **Copy-Konstruktor** auf

---

## Copy-Konstruktor für str

```
str::str(const str &s) {  
    if (s.repr) {  
        repr = new char[strlen(s.repr)+1];  
        strcpy(repr,s.repr);  
    } else {  
        repr = 0;  
    }  
}
```

- Copy-Konstruktor: Parameter ist Referenz auf Klasse selbst
- Achtung: Parameter ist **Referenz** const str & (**nicht** str)
- Wenn der zu kopierende String nicht leer ist, lege neuen Speicher an

⇒ Aliasing aus Beispiel ist aufgelöst

⇒ Ausgabe ist

```
> g++ str2.cpp & ./a.out  
At 1: Still ok  
At 2: Still ok
```

---

# Zerstörung

- Am Ende der Lebenszeit einer Variable wird deren Speicher freigegeben
  - Parameter bei Ende der Funktion
  - Lokale Variablen am Ende des Blocks
  - Temporärer Werte: Nach Gebrauch
  - Heap-Werte: Bei delete
- Hierzu ruft der Compiler den **Destruktor** des Typs auf
- Bei der str-Klasse sollte der Speicher bei repr freigegeben werden

```
class str {  
    public:  
        ~str();  
        ...  
};  
str::~str() {  
    if (repr) delete[] repr;  
}
```

---

## Einschub: Operatorüberladung

- C++ sagt: Operatoren sind nur besondere Funktionen
  - ⇒ Im Abstrakten Syntaxbaum steht: `operator+(i,1)` für Quelltext `i+1`
  - ⇒ Operatoren kann man überladen wie andere Funktionen auch
  - Anwendung: `ostream` überlädt `operator<<` für Ausgabe
  - Mindestens ein Operand muß benutzerdefinierter Typ sein
  - Einschränkung
    - = (Zuweisung)
    - [] (Indizierung)
    - -> (Member-Zugriff)
    - () (Funktionsaufruf)
- müssen als Members von Klassen deklarariert sein
- Linke Seite ist dann das `this`-Argument

---

## Zuweisungsoperator überladen

- Bei Zuweisung wird der alte Wert überschrieben
- ⇒ Die bisher allozierten Ressourcen müssen freigegeben werden
- Eventuell muß neuer Speicher muss angefordert werden
  - Achtung: Selbszuweisung überprüfen

```
void f(A *a, B *b) {  
    *a = *b;  
}
```

- Versteckte Selbszuweisung durch Aliasing
- Bei Ressourcenfreigabe werden Ressourcen der Quelle freigegeben
- Anschließende “Kopie” greift auf freigegebenen Speicher zu

---

## Zuweisung für str

```
class str {
public:
    str &operator=(const str&);
    ...
};
str &str::operator=(const str &s) {
    if (this != &s) {
        if (repr) delete[] repr;
        if (s.repr) {
            repr = new char[strlen(s.repr)+1];
            strcpy(repr,s.repr);
        } else {
            repr = 0;
        }
    }
    return *this;
}
```

---

## Zuweisung vs. Kopie

- Copy-Konstruktor
  - Wird zur Initialisierung verwendet
  - Hat als `this` uninitialisierten Speicher
- Zuweisungsoperator
  - Erhält als `this` vollständig initialisierten Wert
  - Muss Ressourcen von `this` freigeben
  - Muss auf Selbstzuweisung prüfen

---

## Compiler-erzeugte Operatoren

- Bisher: Compiler erzeugt Default Konstruktor, wenn kein Konstruktor angegeben ist
- Dieser Konstruktor initialisiert die member variablen einzeln
- Jetzt: Compiler erzeugt Zuweisung und Copy-Konstruktor
- Diese setzen / kopieren die member variables einzeln

Idiom: Verhindere diese Erzeugung durch `private`: Deklarationen ohne Implementierung.

```
class no_copy {  
    private:  
        no_copy(const no_copy &);  
        no_copy &operator=(const no_copy&);  
}
```

---

## Zusammenfassung: Lebenszyklus in C++

- Der Compiler ruft automatisch Funktionen auf, um Variablen zu verwalten
  - Erzeugung → Konstruktor
  - Kopie → Copy-Konstruktor
  - Zuweisung → operator=
  - Freigabe → Destruktor
- Durch Überladung kann der Programmierer eingreifen

---

## Initialisierung der Member Variables

- Member variablen (Felder) sind auch Variablen
- ⇒ Sie werden durch Konstruktoraufruf initialisiert
- Normalerweise: Default-Konstruktor
- ⇒ Klassenkonstruktor verwendet Zuweisung
- Zuweisung kann für große Objekte ineffizienter sein
- **Initialisierungslisten** rufen Konstruktoren direkt auf

```
str::str() : repr(0) { }
```

- Für die Variablen in der Initialisierungsliste ruft der Compiler keinen anderen Konstruktor auf

---

## Auswertungsreihenfolge

Achtung (!!!): Ausführungsreihenfolge ist Reihenfolge der **Deklarationen**

```
struct A {  
    int i;  
    int j;  
    A() : j(5), i(j) { }  
};  
int main() {  
    A a;  
    cout<<"a.i = "<<a.i<<"", a.j = "<<a.j<<endl;  
}
```

Ergibt

```
a.i = -1077942792, a.j = 5
```

---

## Freigabe und Destruktoren

- Prinzip: Bei Speicherfreigabe werden Destruktoren in der umgekehrten Reihenfolge aufgerufen wie die Konstruktoren beim Anlegen.
    1. Der Destruktor des freigegebenen Wertes selbst
    2. Die Destruktoren der member variables
- ⇒ Fortsetzung der geschachtelten Lebenszeiten für Variablen

---

## Beispiel: Auswertungsreihenfolge

```
class A { ...
public:
    A() : i(next_id++) {
        cout<<"Constructor A"<<i<<endl;
    }
    A(int i) : i(i) {
        cout<<"Constructor A"<<i<<endl;
    }
    ~A() {
        cout<<"Destruktor A"<<i<<endl;
    } };
class B {
private: A a_a, a_b, a_c;
public:
    B() : a_c(3), a_a(1) {
        cout<<"Constructor B"<<endl;
    }
    ~B() {
        cout<<"Destruktor B"<<endl;
    } };
```

```
> ./a.out
Constructor A1
Constructor A100
Constructor A3
Constructor B
Destruktor B
Destruktor A3
Destruktor A100
Destruktor A1
```

---

## const-Werte

- Idee: const-Werte sind nicht veränderbar
- Übertragung auf benutzerdefinierte Typen:
  - Member variables dürfen sich nicht ändern
  - ⇒ Member functions dürfen nicht schreiben
  - ⇒ Man darf für const Werte nicht beliebige Member functions aufrufen
- Lösung: const member functions dürfen Variablen nicht ändern

```
class str {
public:
    char operator[](int i) const;
    ...
};
char str::operator[](int i) const {
    return repr[i];
}
```

---

## Überladung nach const

- Achtung: Das const muss bei Definition mit angegeben werden
- Sogar: Überladung nach const ist möglich:  
Für const-Werte wird eine const member function bevorzugt

```
struct B {  
    void f() const {  
        cout<<"f const"<<endl;  
    }  
    void f() {  
        cout<<"f"<<endl;  
    }  
};  
int main() {  
    B b1;  
    const B b2 = B(); // Initialisierung erforderlich  
    b1.f();  
    b2.f();  
}
```

---

# Konversionen

- Eingebaute Typen bieten Konversionen
- Benutzer-definierte Typen können Konversionen deklarieren
  - Konstruktoren mit einem Argument sind **Konversionskonstruktoren**
  - `operator T()` als member function konvertiert nach  $T$

```
class C {
private:
    int i;
public:
    C(int i) : i(i) { }
    operator int() const {
        return i;
    }
};

C f(C c) {
    return c;
}

...
int r = f(42);
cout<<"Zweimal " <<r<<endl;
```

- Konstruktor ohne Konversion: `struct D { explicit D(int i); };`

---

# Inlining

- Effizienzfrage: Funktionsaufruf
- Beispiel: `operator[]` für `str`-Klasse
  - Eingebauter Typ `char *repr;` → `repr[i]` ist eine Instruktion
  - Mit `str`-Klasse ist `s[i]` ein Funktionsaufruf
- Lösung deklariere Funktionen `inline` (nicht nur member functions)
  - Compiler ersetzt Aufruf durch Rumpf der Funktion
  - Schnell vor allem bei `const` & Argumenten
  - Bei großen Funktionen: Mehr Code
- Zwei Möglichkeiten
  - Rumpf der Methode direkt in der Klasse angeben
  - `inline` vor die Definition stellen
- `inline` ist nur Hinweis, wird erst bei Optimierung (`-O3`) benutzt

---

## C++: Klassen als kleine, konkrete Typen

- C++ Klassen erlauben detaillierte Kontrolle über
  - Anlegen und Initialisierung
  - Kopie
  - Zuweisung
  - Freigabe von Werten
- Programmierer muß viele technische Details beachten
- Durch `inline` member functions können benutzer-definierte Datentypen so effizient sein wie eingebaute
- Durch überladene Operatoren können sie die gleiche knappe Syntax haben wie eingebaute Typen