Objektorientierte Programmierung
(Winter 2008/2009)

Kapitel 12: Überladene Funktionen und Operatoren

- Überladene Funktionsnamen
- Zusammenspiel mit Überschreiben in Subklassen
- Default-Argumente
- Überladen von Operatoren

Überladene Funktionen (1)

- Es kann mehrere Funktionen mit gleichem Namen geben, die sich in der Parameterliste (Anzahl oder Typen der Argumente) unterscheiden.

  Der Spezialfall, daß in mehreren Klassen Funktionen (Methoden) gleichen Namens definiert sind, wurde oben schon betrachtet: Durch den Datentyp (Klasse) des Objektes, auf das die Methode angewendet wird, kann der Compiler entscheiden, welche Funktion er aufrufen soll. Dieses Konzept funktioniert auch allgemeiner.

- Außer im Fall von Default-Argumentwerten (s.u.) haben diese Funktionen nichts mit einander zu tun: Der Programmierer muß für jede Funktion eine eigene Implementierung (Funktionsrumpf) schreiben.

Überladene Funktionen (2)

- Zweck dieses Sprachelementes ist es, semantisch ähnlichen Aktionen auf unterschiedlichen Datentypen den gleichen Namen geben zu können.

  Das ist aber nur Konvention/guter Stil. Theoretisch könnten die Funktionen ganz unterschiedliche Dinge tun.

- Z.B. ist der Ausgabe-Operator << je nach Typ des auszugebenen Wertes verschieden implementiert.

  Man kann auch zur Ausgabe von Objekten eigener Klassen eine Funktion definieren, die dem Operator << hinterlegt ist.

- Überladene Funktionen sind auch für generische Programmierung (Templates, s. Kap. 13) wichtig.

Überladene Funktionen (3)

- Überladene Operatoren gibt es in Programmiersprachen schon länger: Die Addition + ist für ganze Zahlen und für Fließkomma-Zahlen durch völlig unterschiedliche Maschinenbefehle implementiert.

  Früher hatten viele CPUs (z.B. 8086) keine Fließkomma-Operationen eingebaut. Diese Operationen wurden dann durch Aufruf einer Bibliotheksfunktion ausgeführt. Bei Bedarf: Mathematischer Coprozessor.

- Jetzt: Auch für selbst programmierte Funktionen.
- Sonst mußte man Namen wie `print_int`, `print_str` etc. verwenden, oder die Typsicherheit aufgeben.

  Z.B. `printf` in C: Compiler kann Typkorrektheit nicht (immer) prüfen.

Überladene Funktionen (4)

- "Most often, it is a good idea to give different functions different names, but when some functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name."

  [Stroustrup, 2000, Seite 149]

- "Using the same name for operations on different types is called overloading."

  [Stroustrup, 2000, Seite 149]

Überladene Funktionen (5)

- Man kann in C++ also mehrere Funktionen innerhalb einer Klasse oder global definieren, die sich nur in Anzahl oder Typen der Parameter unterscheiden.
- Es dürfen nur nicht exakt die gleichen Parameter-Typen sein, sonst bekommt man eine Fehlermeldung wie "function `int f(int)` already has a body".

  Selbst wenn z.B. `int` und `long` beides 32-Bit Binärzahlen sind, also die gleiche interne Repräsentation haben, sind es doch verschiedene Datentypen. Der Compiler kennt beim Aufruf den exakten Typ des Argumentes, kann also eine der beiden Versionen auswählen.

Überladene Funktionen (6)

- Stimmen die Datentypen der aktuellen Parameter exakt mit den Typen der formalen Parameter überein, so ist klar, welche Funktion aufgerufen wird.

  Bei Bedarf kann man das durch explizite Typ-Umwandlungen erreichen, z.B. macht `static_cast<long>(i)` aus `int i` einen `long`-Wert.

- Komplizierter wird die Funktionsauswahl, wenn es keine exakte Übereinstimung gibt, und die üblichen impliziten Typ-Umwandlungen angewendet werden müssen.

  Für die Bindung eines aktuellen Parameters an einen formalen Parameter gelten die gleichen Regeln wie für eine Zuweisung.

Überladene Funktionen (7)

- Es ist möglich, daß von den gegebenen Typen der aktuellen Parameter durch implizite Umwandlung die Typen der formalen Parameter verschiedener Funktionen erreicht werden können.
- Deswegen gibt es eine Prioritätenliste von einfachen zu komplizierten Typumwandlungen.

  Man kann dies gewissermaßen als Distanz des gegegebenen Typs zum Zieltyp auffassen.

- Es wird die Funktion gewählt, die über einfachere Typumwandlungen erreicht werden kann.

Überladene Funktionen (8)

- Falls die Prioritätenliste keine Entscheidung liefert, ist der Aufruf mehrdeutig und daher unzulässig.

  "f: ambiguous call to overloaded function."

- Dies gilt auch, wenn in verschiedenen Argumentpositionen unterschiedliche Funktionen über einfachere Typ-Umwandlungen erreichbar wären.

  "f: 2 overloads have similar conversions."

- Prinzip: Wenn nicht klar ist, was der Programmierer gemeint hat, sollte der Compiler besser nachfragen.

  Der Programmierer kann es dann durch eine explizite Typumwandlung (s.o.) klar machen.

Überladene Funktionen (9)

Prioritätenliste (Übereinstimmungsgrad):

- Exakt gleicher Typ

  Hierzu zählt auch die Umwandlung von einem Array-Typ in den entsprechenden Pointer-Typ, von einem Funktionsnamen in den entsprechenden Funktions-Pointer-Typ, sowie von T in `const` T.

- Wert-erhaltene Typumwandlungen:
  - ◇ "integral promotions" (s.o.)

    `bool`, `char`, `short` werden in `int` umgewandelt, sowie `unsigned char`, `unsigned short` in `int` (oder `unsigned int`, falls `int` nicht alle Werte des `unsigned`-Typs repräsentieren kann). Aufzählungs-Typen und `wchar_t` werden auch in `int` umgewandelt (oder bei Bedarf in einen größeren Typ: `unsigned int`, `long`, `unsigned long`).
  - ◇ `float` in `double`.

Überladene Funktionen (10)

Prioritätenliste (Übereinstimmungsgrad), Forts.:

- Weitere Standard-Typumwandlungen, z.B.
  - ◇ zwischen `int` und `double` (in beiden Richtungen),
  - ◇ zwischen `int` und `unsigned int` (beide Richt.),
  - ◇ von einem Zeiger auf eine Unterklasse in einen Zeiger auf die Oberklasse,

    Ist `A` Unterklasse von `B`, und `B` Unterklasse von `C`, hat z.B. eine Umwandlung von `A*` nach `B*` Priorität vor der nach `C*`. Alle diese Umwandlungen haben wiederum Priorität vor der nach `void*`.
  - ◇ vom jedem Zeigertyp in `void*`.
- Benutzer-definierte Typumwandlungen (s.u.)
- Funktionen mit variabler Argumentanzahl (s.u.)

Überladene Funktionen (11)

- C++ erlaubt kein Überladen des Ergebnistyps einer Funktion, d.h. es kann nicht zwei Funktionen (in der gleichen Klasse oder global) geben, die sich nur im Resultat-Typ unterscheiden.
- Dadurch kann der Compiler bei komplex geschachtelten Ausdrücken die Funktionsauswahl von innen nach außen durchführen: Jeder Teilausdruck hat einen eindeutig bestimmten Typ.

  Wäre auch das Überladen des Resultattyps möglich, müßten verschiedene Alternativen (z.B. mittels Backtracking) weiter verfolgt werden.

Überladene Funktionen (12)

Aufgabe/Beispiel:

- Gegeben:
  - ◇ `void f(int a) { cout << "f1"; }`
  - ◇ `void f(double a) { cout << "f2"; }`
- Was wird bei folgenden Aufrufen ausgegeben?
  - ◇ `f('a');`
  - ◇ `float x = 1.0; f(x);`
  - ◇ `unsigned int u = 1; f(u);`
  - ◇ `const char* p = "abc"; f(p);`

Überladen&Überschreiben (1)

- Seien folgende Klassendeklarationen gegeben:

```
class C {
    public:
        void f(double x) { cout << "f1:" << x; }
};
class D : public C {
    public:
        void f(int x) { cout << "f2:" << x; }
};
```

- Ruft man nun `d.f(1.7)` auf (für ein Objekt `d` der Klasse `D`), so wird `f2:1` ausgegeben.

Überladen&Überschreiben (2)

- Dies ist etwas überraschend, da die Variante von `f` in `C` vom Typ her viel besser passen würde.
- Die Regel ist einfach:
  - ◇ Der Compiler sucht immer vom Typ des Objektes an in der Klassenhierarchie aufwärts nach einer Funktion mit dem richtigen Namen.
  - ◇ Hat er in einer Klasse eine Funktion mit dem gewünschten Namen gefunden, endet die Suche. Oberklassen werden dann nicht mehr betrachtet.

Überladen&Überschreiben (3)

- Man kann also nur innerhalb der gleichen Klasse einen Funktionsnamen überladen.
- Man kann das Problem aber so lösen:

```
class C {
    public:
        void f(double x) { cout << "f1:" << x; }
};
class D : public C {
    public:
        void f(double x) { C::f(x); }
        void f(int x) { cout << "f2:" << x; }
};
```

Überladen&Überschreiben (4)

Aufgabe/Beispiel:

- Seien folgende Klassendeklarationen gegeben:

```
class C {
    public:
        void g(int n) { cout << "g1:" << n; }
};
class D : public C {
    public:
        void g() { cout << "g2: -"; }
};
```

- Was passiert, wenn man `d.g(0)` aufruft?

  Dabei sei `d` wieder ein Objekt der Klasse `D`.

Überladen&Überschreiben (5)

- Die Regel für virtuelle Funktionen ist anders:

```
class C {
    public:
        virtual void f(double x)
                { cout << "f1:" << x; }
};
class D : public C {
    public:
        void f(int x) { cout << "f2:" << x; }
};
```

- `D d; d.f(1.7);` druckt (wie vorher) `f2:1` aus.
- `D d; C* c = &d; c->f(1.5);` druckt dagegen `f1:1.7`.

Überladen&Überschreiben (6)

- Hier hat die Funktion `f` in der Unterklasse also nicht die Funktion `f` in der Oberklasse überschrieben.
- Tatsächlich steht im Buch von Stroustrup auch, daß, wenn man eine von der Oberklasse ererbte Funktion in der Unterklasse überschreiben will, die Argumenttypen genau passen müssen.

  Das leuchtet ja auch ein. Was oben passiert (syntaktisches Überschreiben statt Überladung) ist schon eher überraschend. Übrigens gibt es für die Regel bei virtuellen Funktionen auch einen technischen Implementierungsgrund: Alle zur Laufzeit möglicherweise ausgewählten Funktionen müssen die gleiche Schnittstelle haben, es wäre sehr umständlich, noch eine bedingte Typumwandlung mit zu generieren.

Überladen&Überschreiben (7)

- Der Resultattyp darf bei der überschreibenen Funktion (in der Unterklasse) auch eine Unterklasse des Resultattyps der ererbten Funktion liefern.

  Die Funktion in der Unterklasse darf den Resultattyp also weiter einschränken. Theoretisch (nicht in C++) wäre es möglich, den Argumenttyp umgekehrt zu erweitern (so wäre die Funktion in der Unterklasse noch kompatibel mit der Funktion in der Oberklasse).

- Es ist interessant (und vermutlich schlecht), daß sich der syntaktische Mechanismus zum Überschreiben (bei bekannter Klassenzugehörigkeit) anders verhält als der dynamische Mechanismus (bei erst zur Laufzeit bekannter Klasse).

Lokale Deklarationen (1)

- Man kann in C++ Funktionen lokal (im Innern einer anderen Funktion) deklarieren.

  Aber nicht definieren, d.h. keinen Funktionsrumpf im Rumpf einer anderen Funktion schreiben.

- Hier gilt die gleiche Regel:
  - ◇ Der Compiler sucht Funktionsdeklarationen von innen nach außen.
  - ◇ Hat er eine Funktion mit dem richtigen Namen gefunden, stoppt er die Suche.

    Das gilt auch, wenn es weiter außen eine Funktion mit einem besser passenden Argumenttyp gibt. Die wird nicht gefunden.

Lokale Deklarationen (2)

- Beispiel:

```
#include <iostream>
using namespace std;
void f(int n) { cout << "f1"; }
void f(double x) { cout << "f2"; }
int main()
{
        void f(double x);
        f(1); // druckt f2
        return 0;
}
```

- Es wird die Version für `double`-Werte aufgerufen, obwohl die andere Version besser passen würde.

Default-Argumente (1)

- Manchmal wird eine Funktion fast immer mit dem gleichen Wert für einen bestimmten Parameter aufgerufen, und nur ganz gelegentlich mit einem anderen Wert.

  Z.B. könnte eine Funktion zum Drucken ganzer Zahlen einen Parameter für die Basis haben. Diese wird fast immer 10 sein, aber gelegentlich auch 16, 2, oder 8.

- Es ist dann nicht schön, daß für ein ganz selten benötigtes Feature jeder Aufruf verkompliziert wird.

  Die meisten Anwender der Druckfunktion wollen nur Dezimalzahlen drucken. Es interessiert sie überhaupt nicht, daß die Funktion die Zahlen auch hexadezimal ausgeben kann.

Default-Argumente (2)

- Lösung mit überladenen Funktionen:

```
void print(int val, int base) { ... }
inline void print(int val) { print(val, 10); }
```

- Da dieser Fall aber offenbar recht häufig vorkommt, bietet C++ ein spezielles Konstrukt dafür an:

```
void print(int val, int base = 10) { ... }
```

- Falls die Funktion `print` mit nur einem Argument aufgerufen wird, fügt der Compiler automatisch den Wert 10 für den zweiten Parameter ein.

Default-Argumente (3)

- Default-Werte sind Werte, die verwendet werden, wenn der Benutzer nicht explizit etwas gesagt hat.

  Deutsch: Voreinstellungen. "Default" heißt u.a. Fehlen, Versäumnis.
- Man kann auch für mehrere Parameter Defaultwerte deklarieren.
- Da aktuelle und formale Parameter über die Position zusammengebracht werden, kann man nur "von hinten" Werte weglassen.

  Wenn eine Funktionen z.B. drei Parameter `A`, `B`, `C` hat, und für alle drei Parameter Default-Werte deklariert sind, kann man sie mit null, einem (`A`), zwei (`A`, `B`) und drei Argumenten (`A`, `B`, `C`) aufrufen.

Default-Argumente (4)

- Sind Deklaration (nur Kopf) und Definition (mit Rumpf) der Funktion von einander getrennt, so gilt:
  - ◇ Die Default-Argumente müssen in der Deklaration der Funktion angegeben werden, und
  - ◇ dürfen in der Definition nicht wiederholt werden.

    Genauer darf man in einem "Scope" (Geltungsbereich, Block) nur einmal Defaultwerte angeben. Das könnte auch in der Definition sein, muß aber natürlich vor dem Aufruf geschehen.
- Der Grund ist, daß die Defaultwerte vom Compiler beim Funktionsaufruf eingesetzt werden.

  Es gibt hier nur einen Funktionsrumpf. Dies ist ein Unterschied zu überladenen Funktionen, wo es mehrere Funktionsrümpfe gibt.

Variable Argumentanzahl (1)

- In C gab es kein Überladen / keine Default-Werte, aber man konnte Funktionen mit variabler Argumentanzahl definieren.
- Wichtigste Anwendung war die Funktion `printf` zur Ausgabe:

```
int printf(const char* format ...);
```

Im Format-String, der als erstes Argument übergeben wurde, haben Format-Elemente wie `%d` (Dezimalzahl) und `%s` (String) auf weitere Argumente hingewiesen, die an der betreffenden Stelle in die Ausgabe eingefügt werden sollten. Z.B. druckt `printf("%s = %d!\n", "n", 5);` die Ausgabe "`n = 5!`".

Variable Argumentanzahl (2)

- Man kann diesen Mechanismus auch in C++ noch verwenden, aber der Compiler prüft dann die Typen der Argumente nicht.

  Die Ausgabefunktion war ein Überbleibsel aus der Zeit, als in C die Argumenttypen nicht geprüft wurden (außer mit Zusatzwerkzeugen wie `lint`). Später haben dann bessere Compiler bei explizit angegebenen Formatstrings Warnungen ausgegeben, wenn die Typen der weiteren Argumente nicht passten. Man kann den Formatstring aber auch im Programm berechnen, dann wird die Typkorrektheit unentscheidbar.
- Da die Typsicherheit nicht gewährleistet werden kann, sollte man auf dieses Konstrukt verzichten.

  Notfalls schaue man in die Include-Datei `cstdarg` und da die Makros `va_list`, `va_start`, `va_arg`, `va_end`.

Überladene Operatoren (1)

- Man kann in C++ die Operatoren der Sprache (wie +, ∗, etc.) auch für eigene Datentypen definieren.

  D.h. das syntaktische Symbol mit einer weiteren Funktion überladen. Die Funktionsauswahl geschieht über die Datentypen der Argumente.

- Z.B. wurde für die Bibliotheksklasse `string` definiert, daß + die String-Konkatenation bezeichnet.

  Bibliotheksklassen und Bibliotheksfunktionen sind nicht in die Sprache eingebaut, sondern verwenden nur Sprachkonstrukte, die man auch für eigenen Programmcode benutzen kann. Im allgemeinen ist es gut, wenn die Sprache relativ klein ist, und möglichst viel als Bibliotheksfunktion realisiert ist (mit einsehbarem und ggf. änderbarem Quellcode). Das spricht für die Möglichkeit, Operatoren benutzer-definierbar zu machen. Leider ist C++ aber dennoch eine komplexe Sprache.

Überladene Operatoren (2)

- Z.B. würde der Operator + Sinn machen für rationale und komplexe Zahlen, Vektoren, Datums-Werte (mit Zeitspanne als zweitem Argument), etc.

  Es können nicht alle diese Typen in die Sprache selbst eingebaut sein.

- Man kann z.B. den Operator << auch zur Ausgabe eigener Typen verwenden.

  Er sollte für alle Typen definiert sein.

- "It is hard to overestimate the importance of concise notation for common operations."

  [Stroustrup, 2000, Seite 261]

Überladene Operatoren (3)

Beispiel:

```
class Date {
    private:
        int day; int month; int year;
    public:
        Date(int d, int m, int y) { ... }
        int get_day() const { return day; }
        ...
        Date operator +(int diff_days) {
                Date tmp(day, month, year);
                ... // add diff_days to tmp
                return tmp;
        }
};
```


Überladene Operatoren (4)

- Man kann einen Operator `x` definieren, indem man eine Funktion mit dem Namen `operator x` definiert.

  "`operator`" ist ein Schlüsselwort. Das Leerzeichen vor `x` ist nicht nötig.

- Das funktioniert nicht nur für "member functions" (Methoden), sondern auch für normale Funktionen:

```
ostream& operator<<(ostream& o, const Date& d)
{
        o << d.get_day() << '.';
        o << d.get_month() << '.';
        o << d.get_year();
        return o;
}
```

Überladene Operatoren (5)

- Anschließend kann man z.B. folgendes schreiben:

```
Date x(15, 12, 2006);
Date y = x + 9;       // ruft operator+ auf
cout << y;            // ruft operator<< auf
```

- Man kann in C++ keine grundsätzlich neuen Operatoren definieren.

  Man kann nur die vorhandenen Operatoren für neue Datentypen überladen. Die kontextfreie Syntax der Sprache ändert sich dadurch nicht. In der Sprache Prolog kann man eigene Operatoren definieren.

- Man kann in C++ nicht die Bedeutung der Operatoren für die eingebauten Datentypen ändern.