## Vorlesung

# Objektorientiertes Programmieren in C++

## Teil 5 - WS 2025/26

**Detlef Wilkening**
**www.wilkening-online.de**

# 11 Typ-System und mehr

## 11.1 Typ-Aliase

### 11.1.1 Motivation

Spätestens seit den Containern sollte Ihnen klar geworden sein, dass das C++ Typ-System komplizierte und lange Typen ermöglicht.

```
Map:              std::map<int, std::string>
Wert:             std::map<int, std::string>::value_type
Const-Iterator:   std::map<int, std::string>::const_iterator
```

Oder stellen Sie sich vor, Sie benutzen als Schlüssel einen String, und als Wert einen Vektor von Strings:

```
Map:              std::map<std::string, std::vector<string>>
Wert:             std::map<std::string, std::vector<string>>::value_type
Const-Iterator:   std::map<std::string, std::vector<string>>::const_iterator
```

Wie man sieht, erlaubt C++ sehr lange Typ-Namen – und dies ist erst der Anfang. Von daher wäre es natürlich schön, für diese Typen kürzere – und vielleicht auch bessere, da sprechendere – Namen vergeben zu können.

### 11.1.2 Using

Mit dem Schlüsselwort „using“ können neue Typ-Namen definert werden, sogenannte Typ-Aliase. Achtung - keine neuen Typen, sondern nur neue Namen.

```
using length = int;       // Definiert einen neuen Namen length fuer den Typ int
length len = 17;          // Achtung - len ist weiter vom Typ int!
```

Wozu werden Typ-Aliase benutzt?

1. Um verständliche Typ-Namen bei komplexen oder unverständlichen Typen zu bekommen.
2. Um einen konkreten Typ verstecken zu können.
3. Um semantische Typ-Namen zu erzeugen.
4. Um einen Typ leicht verändern zu können.

#### 11.1.2.1 Verständliche Typ-Namen bei komplexen Typen

Dies sollte sofort klar sein, denken Sie an unser Map Beispiel mit value_type:

```
using container = map<int, string>;
using value = container::value_type;
using iter  = container::const_iterator;

container m;
m.insert(value(2, "Heinrich"));
m.insert(value(3, "Ede"));
m.insert(value(1, "Ansgar"));
for (iter i=begin(m); i!=end(m); ++i)
{
   cout << i->first << ' ' << i->second << '\n';
}
```

```
Ausgabe
1 Ansgar
2 Heinrich
3 Ede
```

#### 11.1.2.2 Verstecken eines konkreten Typ's

Was glauben Sie wohl, wie die Typen „std::streamsize“ und „std::string::size_type“ entstehen? Natürlich sind es Typ-Aliase auf die passenden elementaren Datentypen. Immer dann, wenn Sie – aus welchen Gründen auch immer – den konkreten Typ verstecken wollen ist ein Typ-Alias das Mittel der Wahl.

#### 11.1.2.3 Semantische Typ-Namen

Hand in Hand mit dem Verstecken des konkreten Typ's geht die Möglichkeit, dem Benutzer einen Typ-Namen an die Hand zu geben, der ihm etwas sagt.

**Beispiel:**

Folgende Funktion soll wohl ein Rechteckt zeichnen.

```
void draw_rect(int, int, int, int);
```

Während die reine Funktionalität noch recht offensichtlich ist, stellt sich die Frage, wofür die einzelnen Parameter sind. Es gibt prinzipiell viele Möglichkeiten, aber im Rahmen der normalen Erwartungshaltung (z.B. X-Werte stehen vor Y-Werten), sind zwei Varianten recht wahrscheinlich:

```
void draw_rect(int x1, int y1, int x2, int y2);
```

```
void draw_rect(int x, int y, int width, int height);
```

Daher sind die Parameter:

- entweder die x/y-Koordinaten der linken/oberen und der rechten/unteren Ecke,
- oder die x/y-Koordinaten der linken/oberen Ecke, und Breite und Höhe.

Aber welche Interpretation ist denn jetzt die richtige?

Hier z.B. könnten semantische Typ-Namen helfen:

```
using xcoor = int;
using ycoor = int;
void draw_rect(xcoor, ycoor, xcoor, ycoor);
```

Ah, es muss sich wohl um die x/y-Koordinaten der linken/oberen und der rechten/unteren Ecke handeln.

#### 11.1.2.4 Leichte Veränderung eines Typ's

Wenn sie den eigentlichen Typ durch ein Typedef im Code mit einem Typ-Alias versehen, und im Code nur den Typ-Alias verwenden, so können sie den Typ sehr schnell und leicht austauschen.

Vorher:

```
using age = int;
```

Nachher:

```
using age = short;
```

Wenn z.B. der Typ „int" nicht optimal ist (vielleicht ist er größer als notwendig, und der zusätzliche Speicherbedarf ist relevant), so kann er mit einer Änderung verändert werden.

#### 11.1.2.5 Alternative „typedef"

Seit C++11 können Typ-Aliase mit „using" definiert werden. Alternativ können sie auch mit „typedef" definiert werden – dies ist eine alte von C geerbte Syntax. Mit „using" definierte Typ-Aliase sind aber einfacher zu Schreiben und zu Lesen, und haben mehr Möglichkeiten – Sie sollten also bevorzugt werden.

## 11.2 Referenzen

Eine Referenz ist ein alternativer Name für ein Objekt.

Auch im täglichen Leben haben viele Dinge mehrere Namen:
Auto: *mein Auto*, *Zorro*, *alte Schrottkarre*.
Mann: *Schnuckiputzi*, Herr Müller, *mein Gatte*, *mein Liebster, der Olle*.

Obwohl verschiedene Namen benutzt werden, ist immer das gleiche Objekt gemeint. Man könnte alle diese Namen als Verweise oder Referenzen auf das Original-Objekt bezeichnen.

Daher ist es vollkommen egal, ob sie auf ein Objekt über die Original-Variable oder eine mit ihr assoziierte Referenz zugreifen – vom Verhalten sind beide Varianten **absolut äquivalent.**

Eine Referenz wird definiert, in dem zwischen den Typ und den Variablennamen (hier auch Referenznamen) das „Kaufmanns-Und“ „&“ eingefügt wird.

„Typ&“ bedeutet Referenz auf Typ

```
int i=17;                                   // Integer-Variable i
int& r1=i;                                  // Referenz r1 als zweiter Name zu i
int& r2=i;                                  // "         r2 als dritter "    "  i

cout << i << ' ' << r1 << ' ' << r2 << '\n';     // Ausgabe: 17 17 17

++i;
cout << i << ' ' << r1 << ' ' << r2 << '\n';     // Ausgabe: 18 18 18

r1=-4;
cout << i << ' ' << r1 << ' ' << r2 << '\n';     // Ausgabe: -4 -4 -4

r2+=32;
cout << i << ' ' << r1 << ' ' << r2 << '\n';     // Ausgabe: 28 28 28
```

Alle drei Variablen mit den Namen *i*, *r1* und *r2* verweisen auf die gleiche Variable. Egal, welcher Name benutzt wird, alle Operationen wirken auf dieselbe Variable.

**Eine Referenz ist völlig gleichbedeutend zum Original-Objekt.**

Die erlaubten Operationen auf eine Referenz sind somit die erlaubten Operationen auf das Original-Objekt. Sie betreffen nie die Referenz, sondern immer das Original-Objekt. Daher kann eine Referenz nicht aufgelöst, bearbeitet und verändert werden - alle Operationen wirken auf das referenzierte Objekt und nicht die Referenz. **Eine Referenz verweist ihr gesamtes Leben auf das gleiche Objekt.**

**Konsequenz** für die Initialisierung: da eine Referenz als Synonym für ein Objekt steht, und jede Operation auf das Original-Objekt wirkt, muss sie bei der Definition sofort initialisiert werden.

```
double& rd;    // Compiler-Fehler - eine Referenz muss initialisiert werden
```

**Hinweis** – seit C++11 gibt es zusätzlich sogenannte R-Value Referenzen. Die hier bislang besprochenen „alten“ Referenzen gab es schon in C++98, und werden jetzt zur Unterscheidung korrekt L-Value Referenzen genannt. Wenn Sie in einem C++ Dokument den Kurz-Begriff „Referenz“ finden, so ist fast immer eine L-Value Referenz gemeint.

### 11.2.1 Const-Referenzen

Um Konstanten zu referenzieren, gibt es auch „const“-Referenzen. Und auch über „const“-Referenzen können Objekte natürlich nicht verändert werden.

```
const int width = 65;
const int& rw = width;

++rw;       // Compiler-Fehler, rw ist const
```

Um vielfältige Kombinationen in der Benutzung zu erlauben, können „const“-Referenzen auch non-const Objekte referenzieren, d.h. das Zugriffsrecht kann eingeschränkt werden.

```
int sum = 89;
const int& rsum = sum;

++sum;      // natuerlich okay
++rsum;     // Compiler-Fehler, rsum ist const
```

Umgekehrt darf das Zugriffsrecht über Referenzen natürlich nie erweitert werden, d.h. non-const Referenzen auf Konstanten sind nicht erlaubt.

```
const int ci = 17;
int& ri = ci;            // Compiler-Fehler - const wuerde unterlaufen werden
```

### 11.2.2 Referenzen mit „auto“

Auch Referenzen und Const-Referenzen können mit der Auto Typ-Deduktion von C++ kombiniert werden. Analog zu „const“ mit „auto“ muss man hinter dem Schlüsselwort „auto“ das Referenz-Symbol „&“ angeben – dann deduziert der Compiler den Initial-Variablen-Typ und erzeugt eine Referenz bzw. Const-Referenz darauf.

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

int main()
{
   int n = 16;
   auto& rn = n;         // Erzeugt eine Int-Referenz, kein Int, da "auto" mit Referenz

   cout << "Vor  ++rn - n:  " << n << '\n';    // -> Inital-Wert 16
   ++rn;
   cout << "Nach ++rn - n:  " << n << '\n';    // Ausgabe von 17, da "rn" Referenz ist

   const auto& crn = n;                        // Erzeugt eine Const-Int-Referenz

   cout << "Vor  ++n - crn: " << n << '\n';    // -> Aktueller Wert 17
   ++n;
   cout << "Nach ++n - crn: " << n << '\n';    // Ausgabe von 18, da "crn" Referenz ist
}
```

**Ausgabe**

```
Vor  ++rn - n:  16
Nach ++rn - n:  17
Vor  ++n - crn: 17
Nach ++n - crn: 18
```

### 11.2.3 Referenzen und Container

Aufgrund der Semantik von Referenzen beim Erzeugen (sie müssen initialisiert werden), Kopieren und Zuweisen (es werden nicht die Referenzen kopiert bzw. zugewiesen, sondern die referenzierten Objekte), können Container keine Referenzen aufnehmen.

```
vector<int&> v;   // Compiler-Fehler - Referenzen in Containern sind nicht erlaubt
list<int&> l;     // Compiler-Fehler - Referenzen in Containern sind nicht erlaubt
set<int&> s;      // Compiler-Fehler - Referenzen in Containern sind nicht erlaubt
```

### 11.2.4 Referenzen vermeiden Kopien

Referenzen sind in C++ an ganz vielen Stellen extrem wichtig, u.a. da sie häufig Kopien vermeiden. Statt einer Kopie (die Performance und Speicher kostet) ermöglichen es uns Referenzen mit dem Original-Objekt zu arbeiten, und mit Const-Referenzen sogar ohne Gefahr der Veränderung des Originals. Die wichtigste und auch typischte Nutzung von Referenzen sind daher Funktions-Parameter und Funktions-Rückgaben.

Aber auch schon bei z.B. Schleifen-Variablen für die Objekte eines Containers während des Schleifen-Durchlaufs können Referenzen und Const-Referenzen sehr hilfreich sein. Erinnern wir uns z.B. an die range-basierte For-Schleife:

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

int main()
{
   vector<string> v { "Eins", "Zwei", "Drei" };

   for (string s : v)                        // (*)
   {
      cout << s << ' ';
   }
   cout << '\n';
}
```

**Ausgabe**
```
Eins Zwei Drei
```

Die Schleifen-Variable „s“ in Zeile (*) ist keine Referenz – daher bei jedem Durchlauf wird eine Kopie des Objekts im Container erzeugt. Man kann das schön sehen, wenn man zweimal über den Container läuft, und beim ersten Durchlauf „s“ ändert. Im Container ist nichts von der Änderung angekommen.

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

int main()
{
   vector<string> v { "Eins", "Zwei", "Drei" };

   for (string s : v)
   {
      s += "XXX";           // Aenderung ist nur in der lokalen "s" Variable
   }

   for (string s : v)
   {
      cout << s << ' ';     // Ausgabe der initialen Werte im Container
   }
   cout << '\n';
}
```

**Ausgabe**
```
Eins Zwei Drei
```

Ganz anders, wenn wir mit Referenzen arbeiten:

```
#include <iostream>
```

```
#include <string>
#include <vector>
using namespace std;

int main()
{
   vector<string> v { "Eins", "Zwei", "Drei" };

   for (string& s : v)       // Referenz!   (*)
   {
      s += "XXX";            // Aenderung bezieht sich jetzt auf den String im Container
   }

   for (string s : v)        // (**)
   {
      cout << s << ' ';      // Ausgabe der geaenderten Werte im Container
   }
   cout << '\n';
}
```

**Ausgabe**
```
EinsXXX ZweiXXX DreiXXX
```

Die Referenz in Zeile (*) vermeidet Kopien (meist bessere Performance und meist weniger Speicherbedarf) und erlaubt uns hier zusätzlich die Modifikation der Original-Strings im Vektor. Auch für Zeile (**) bietet sich die Referenz an, aber diesmal in der Const-Variante. Denn durch die Referenz gewinnen wir Performance und sparen Speicherplatz – da wir den Original-String im Container aber nicht ändern wollen sichern wir uns durch „const“ ab.

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

int main()
{
   vector<string> v { "Eins", "Zwei", "Drei" };

   for (string& s : v)
   {
      s += "XXX";
   }

   for (const string& s : v)       // Const-Referenz wegen Performance und Speicher
   {
      cout << s << ' ';
   }
   cout << '\n';
}
```

**Ausgabe**
```
EinsXXX ZweiXXX DreiXXX
```

Und bitte – selbst wenn ich dies jetzt mit der range-basierten For-Schleife erklärt habe – das trifft natürlich auch auf unsere normalen Schleifen mit Iteratoren zu:

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

int main()
{
   vector<string> v { "Eins", "Zwei", "Drei" };

   for (vector<string>::iterator it=begin(v); it!=end(v); ++it)
   {
      string s = *it;                              // Kopie - Aenderungen nur lokal
      s += "XXX";
```

```
    }

    for (vector<string>::const_iterator it=begin(v); it!=end(v); ++it)
    {
        cout << *it << ' ';
    }
    cout << '\n';
}
```

**Ausgabe**
```
Eins Zwei Drei
```

Und auch hier wird es anders, wenn wir mit einer Referenz arbeiten:

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

int main()
{
    vector<string> v { "Eins", "Zwei", "Drei" };

    for (vector<string>::iterator it=begin(v); it!=end(v); ++it)
    {
        string& s = *it;                          // Referenz - Aenderung im Container
        s += "XXX";
    }

    for (vector<string>::const iterator it=begin(v); it!=end(v); ++it)
    {
        cout << *it << ' ';
    }
    cout << '\n';
}
```

**Ausgabe**
```
EinsXXX ZweiXXX DreiXXX
```

## 11.3 Aufzählungs-Typen

Manchmal wird einfach ein Typ benötigt, der nur bestimmte Werte annehmen kann.
Beispiele:

- Alignment-Werte: right, center, left, block
- Farb-Werte: blue, red, green,...
- Spielfarben-Werte: white, black
- Spezielle Tasten-Werte: alt, ctrl, shift, meta, ...

Für einen Typ zur Auflistung von Werten gibt es in C++ Aufzählungs-Typen bzw. Enums. Mit der Definition eines Enums erzeugt man einen eigenständigen benutzerdefinierten Typ, der nur die definierten Werte annehmen kann. Eingeleitet wird die Enum-Definition mit den Schlüsselwörtern „enum class“ – dann folgt der Enum-Name, der den normalen C++ Namensregeln genügen muss. Danach folgt in geschweiften Klammern die Auflistung der erlaubten Werte.

```
enum class alignment { right, center, left, block };

enum class game_color { white, black };

enum class special_keys { shift, ctrl, alt, meta };
```

Ein Enum ist ein benutzerdefinierter Daten-Typ, den man wie jeden anderen Typ nutzen kann – also z.B. um Variablen anzulegen oder ihn als Funktions-Parameter oder –Rückgabe nutzen. Die Besonderheit ist, dass man als Werte nur die zum Enum gehörigen Enum-Werte nutzen kann. Die Werte werden dabei über den Enum-Namen mit dem Scope-Resolution-Operator „::“ und dem Enum-Konstanten-Namen angesprochen:

```
enum class alignment { right, center, left, block };

void fct(alignment);     // Funktions-Deklaration mit Enum-Typ - bekommen wir noch

int main()
{
   alignment a = alignment::center;     // Okay - "a" Variable vom Typ "alginment"
   a = alignment::block;                // Okay - Zuweisungen gehen natuerlich auch

   if (a == alignment::block)           // Okay - Vergleiche sind moeglich
   {
   }

   fct(a);                              // Okay
   fct(alignment::left);                // Okay
}
```

Details:

- Die Enum-Werte sind Compile-Zeit-Konstanten, und werden auch Enum-Konstanten oder Aufzählungs-Konstanten genannt.
- Die Aufzählungs-Konstanten werden intern immer durch einen integralen Wert dargestellt. Wenn nicht anders definiert, werden für die Aufzählungs-Konstanten aufsteigende Werte von ‚0’ an vergeben.
- Enum-Variablen und Enum-Konstanten können explizit mit „static_cast“ in einen integralen Wert gewandelt werden – damit können sie z.B. auf „cout“ ausgegeben werden. Eine Alternative für die Ausgabe ist das Schreiben eines eigenen Ausgabe-Operators „<<“ – siehe späteres Kapitel.

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

 int main()
{
   enum class alignment { right, center, left, block };

   cout << "right:  " << static_cast<int>(alignment::right)  << '\n';       // -> 0
   cout << "center: " << static_cast<int>(alignment::center) << '\n';       // -> 1
   cout << "left:   " << static_cast<int>(alignment::left)   << '\n';       // -> 2
   cout << "block:  " << static_cast<int>(alignment::block)  << '\n';       // -> 3

   alignment a = alignment::center;
   cout << "a: " << static_cast<int>(a) << '\n';                // -> 1

   int n = static_cast<int>(alignment::left);
   cout << "n: " << n << '\n';                 // -> 2
}
```

**Ausgabe**

```
right:  0
center: 1
left:   2
block:  3
a: 1
n: 2
```

Es können den Aufzählungs-Konstanten eigene Werte zugewiesen werden, indem die Werte

in der Enum-Definition den Aufzählungs-Konstanten zugewiesen werden. Die Werte müssen dabei integrale Compile-Zeit-Konstanten sein. Wird einer Enum-Konstanten kein Wert zugewiesen, so wird der vorherige Enum-Wert um „1“ erhöht und genutzt. Hierbei dürfen auch mehrere Enum-Konstanten eines Enums den gleichen Wert annehmen.

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

int main()
{
   enum class align { right, center, left, block };

   cout << "right:  " << static_cast<int>(align::right)  << '\n';       // -> 0
   cout << "center: " << static_cast<int>(align::center) << '\n';       // -> 1
   cout << "left:   " << static_cast<int>(align::left)   << '\n';       // -> 2
   cout << "block:  " << static_cast<int>(align::block)  << '\n';       // -> 3

   cout << '\n';

   enum class my_enum
   {
      val1 = 80,                                        // val1 ⇔ 80
      val2 = 90,                                        // val2 ⇔ 90
      val3,                                             // val3 ⇔ 91
      val4 = static_cast<int>(align::left),             // val4 ⇔  2
      val5,                                             // val5 ⇔  3
      val6 = 2,                                         // val6 ⇔  2
      val7                                              // val7 ⇔  3
   };

   cout << "val1: " << static_cast<int>(my_enum::val1)  << '\n';        // -> 80
   cout << "val2: " << static_cast<int>(my_enum::val2)  << '\n';        // -> 90
   cout << "val3: " << static_cast<int>(my_enum::val3)  << '\n';        // -> 91
   cout << "val4: " << static_cast<int>(my_enum::val4)  << '\n';        // -> 2
   cout << "val5: " << static_cast<int>(my_enum::val5)  << '\n';        // -> 3
   cout << "val6: " << static_cast<int>(my_enum::val6)  << '\n';        // -> 2
   cout << "val6: " << static_cast<int>(my_enum::val7)  << '\n';        // -> 3
}
```

**Ausgabe**

```
right:  0
center: 1
left:   2
block:  3

val1: 80
val2: 90
val3: 91
val4: 2
val5: 3
val6: 2
val7: 3
```

Häufig werden Enums in Verbindung mit einer Switch-Anweisung genutzt.

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

enum class key { shift, ctrl, alt };

int main()
{
   key k = key::shift;

   switch (k)
   {
   case key::shift:
      cout << "Taste: shift\n";
      break;
   case key::ctrl:
```

```
      cout << "Taste: ctrl\n";
      break;
   case key::alt:
      cout << "Taste: alt\n";
      break;
   default:
      cout << "Fehler - unbekannte Taste\n";
   }
}
```

```
Ausgabe
Taste: shift
```

Enums gab es schon in C und damit auch in C++98. Sie sind in C++11 aber erweitert worden. Die hier vorgestellten Enums entsprechen den neuen C++11 Enums. Die alten „unscoped Enums“ sollten Sie nicht mehr verwenden.

## 11.4 Typ-Konvertierungen

C++ hat ein sehr kompliziertes und aufwändiges Typ-System, das auf der anderen Seite aber auch sehr leistungsfähig ist. Leider hat das Typ-System von C++ auch einige Mankos, die aus der Historie von C entstanden sind, und in der Praxis immer wieder für Probleme sorgen.

### 11.4.1 Implizite Typ-Konvertierungen

So können z.B. beliebige elementare Datentypen einfach automatisch ineinander umgewandelt werden, ohne das der Benutzer das speziell ausdrücken muss, bzw. es vielleicht auch gar nicht bewußt wahrnimmt. Diese Wandlungen werden „implizite Wandlungen“ bzw. „implizite Typ-Konvertierungen“ genannt, da sie ohne explizite Anweisungen im Quelltext automatisch vom Compiler vorgenommen werden.

```
double d = 3.14;
int i = d;              // Implizite Typ-Konvertierung von "double" => "int"
float f = true;         // Implizite Typ-Konvertierung von "bool" => "float"
char c = f;             // Implizite Typ-Konvertierung von "float" => "char"
long l = 'A';           // Implizite Typ-Konvertierung von "char" => "long"
```

Auch Wandlungen zwischen elementaren Datentypen, die mit Daten-Verlusten behaftet sind (z.B. „double“ zu „int“ – hier gehen alle Nachkomma-Stellen verloren), sind korrekter Code und **müssen** vom Compiler anstandslos compiliert werden. Viele Compiler warnen diese Stellen an – aber es ist korrekter Code, daher muss er anstandslos compiliert werden. Und auch nicht jeder Compiler warnt solche Stellen an.

Implizite Typ-Konvertierungen können in C++ an vielen Stellen vorkommen – nicht nur bei Initialisierungen oder Zuweisungen. Typen können z.B. auch innerhalb von Ausdrücken implizit konvertiert werden – z.B. wenn Sie eine „int“ und eine „unsigned int“ Variable addieren oder vergleichen, der Compiler wählt hier noch genauen Regeln einen gemeinsamen Typ in den er die Variablen implizit konvertiert.

Eine weitere typische Situation für implizite Typ-Konvertierungen sind Funktions-Aufrufe – vergleiche auch Kapitel 12.4. Bei Funktions-Aufrufen können sowohl die Aufruf-Argumente

als auch die Rückgabe implizit konvertiert werden:

```
void fct(int);            // Dies deklariert eine Funktion, die einen "int" erwartet

fct(3.14);                // Ruft mit einem "double" die Funktion "fct(int)" auf
                          // Dabei wird das "double" Argument in einen "int" gecastet
                          // Implizite Typ-Konvertierung mit Daten-Verlust
```

**Hinweis** – der Compiler darf pro impliziter Konvertierungen immer beliebig viele von der Sprache definierte Typ-Umwandlungen, und max. eine benutzerdefinierte Typ-Umwandlung vornehmen. Benutzerdefinierte Typ-Umwandlungen werden wir in Form von Konvertierungs-Konstruktoren kennenlernen. Zusätzlich gibt es noch benutzerdefinierte Konvertierungs-Operatoren, die ein Teil des Themas Operator-Überladung sind – in der Vorlesung aber nicht besprochen werden.

**Hinweis** – Typ-Konvertierungen (sogenannte Casts) sind häufig problematisch, da sie mit Informations-Verlust einhergehen (z.B. „double“ => „int“) und sollten daher möglichst vermieden werden. Die beste und einfachste Lösung besteht darin, wenn möglich, von Anfang an den korrekten Typ zu wählen und ohne Typ-Konvertierungen auszukommen.

**Hinweis** – alle impliziten Konvertierungen sind Teil einer Konvertierungs-Hierarchie, die in Kapitel 12.6.2 grob eingeführt wird. Diese Konvertierungs-Hierarchie spielt eine Rolle, wenn parallel mehrere implizite Konvertierungen möglich sind, wie es z.B. bei der Funktions-Überladung passieren kann – siehe Kapitel 12.6.

### 11.4.2 Explizite Typ-Konvertierungen

Manchmal geht es einfach nicht ohne explizite Typ-Konvertierungen. Folgende Möglichkeiten zur expliziten Typ-Konvertierung sind in C++ vorhanden:

- Die aus C geerbte C-Konvertierung
  Syntax: `(ziel-typ)quell-ausdruck`
- Die Funktionale-Konvertierung, die prinzipiell gleichwertig zur C-Konvertierung ist, aber mit mehr als einem Quell-Argument umgehen kann.
  Syntax: `ziel-typ(quell-ausdruck)`
- Der Operator „static_cast“
  Syntax: `static_cast<ziel-typ>(quell-ausdruck)`
- Der Operator „const_cast“
  Syntax: `const_cast<ziel-typ>(quell-ausdruck)`
- Der Operator „reinterpret_cast“
  Syntax: `reinterpret_cast<ziel-typ>(quell-ausdruck)`
- Der Operator „dynamic_cast“
  Syntax: `dynamic_cast<ziel-typ>(quell-ausdruck)`

Der Operator „dynamic_cast“ hat hierbei eine Sonderbedeutung, da er die einzige Konvertierung darstellt, die zur Laufzeit überprüft und durchgeführt wird. Man kann ihn nur bei Casts in Vererbungs-Hierarchien nutzen, und dort wird er dann auch detailierter vorgestellt.

Die C-Konvertierung und die neue Funktionale-Konvertierung sind prinzipiell gleichwertig, und stellen erstmal nur zwei unterschiedliche Syntaxen dar – auf die genauen Unterschiede werden wir hier nicht eingehen. Beide Konvertierungs-Arten können sehr viele Wandlungen ausführen und entsprechen in ihren Möglichkeiten ungefähr der Summe aus den Operatoren „static_cast“, „const_cast“ und „reinterpret_cast“. Trotzdem ist ihre Nutzung nicht empfehlenswert, da man aufgrund ihrer Mächtigkeit leicht fehlerhafte Casts durchführt, und die Casts im Quelltext auch nicht besonders *auffallen*. Bei der Nutzung der Operatoren „static_cast“, „const_cast“ und „reinterpret_cast“ muss der Nutzer klar hinschreiben, welche Casts er ausführen will, und ein Leser kann die Casts auch leicht erkennen.

**Hinweis** – die Operatoren „const_cast“ und „reinterpret_cast“ werden in der Vorlesung nicht näher besprochen. Der Operator „static_cast“ wird im nächsten Kapitel 11.4.3 vorgestellt, der Operator „dynamic_cast“ später.

## 11.4.3 Konvertierungs-Operator „static_cast“

Der Operator „**static_cast**“ konvertiert zur Compilezeit u.a. folgende Typen ineinander:

- All die, die auch mit den normalen Sprachmitteln ineinander umwandelbar sind
- Integrale Werte können in enums gewandelt werden, und umgekehrt
- Und weitere, die wir zurzeit noch nicht kennen, und auch in der Vorlesung nicht kennenlernen werden

### Syntax

```
static_cast<ziel-typ>(quell-ausdruck)
```

### Beispiele

```
char c = static_cast<char>(65);      // Okay, wuerde aber auch ohne "static_cast" gehen
int i = static_cast<int>(3.14);      // Okay, wuerde aber auch ohne "static_cast" gehen

enum class color { red, blue, green };
color col = static_cast<color>(1);   // Okay, benoetigt zwingend "static_cast"
int v = static_cast<int>(col);       // Okay, benoetigt zwingend "static_cast"
```

**Hinweis** – der Operator „static_cast“ kann noch mehr Typen ineinander umwandeln, diese Konvertierungen sind im Rahmen der Vorlesung aber nicht relevant.

## 11.4.4 Typ-Verwendung

Welchen Typ verwende ich nun für welche Aufgabe?

### 11.4.4.1 Einfache Fälle

Wenn Sie frei wählen können, und keine Abhängigkeiten haben, dann nehmen Sie den elementaren Daten-Typ, der am besten auf Ihre Anforderungen passt.

Häufig haben Sie aber Abhängigkeiten zu anderen Typen. Ihre Variable könnte eine

Abhängigkeit zu den Längen von Strings haben, also z.B. die Länge eines Strings aufnehmen können müssen. Dann ist es sinnvoll, den jeweils von der Bibliothek vorgegebenen Daten-Typ zu nehmen – hier also z.B. „std::string::size_type“. Damit sind sie immer auf der sicheren Seite. Hierbei empfiehlt es sich häufig, zur besseren Lesbarkeit und zum besseren Verständnis ein Typedef für den Typ einzusetzen.

In den meisten Fällen wählen Sie über obige Regel einen Typ aus, und bleiben innerhalb Ihrer Aufgabe auch immer innerhalb des Typs. Das ist das Beste, was Ihnen passieren kann.

# 12 Funktionen

Die in diesem Kapitel vorgestellten Funktionen sind sogenannte „freie Funktionen“, da sie keinem speziellen Kontext zugeordnet sind. Zusätzlich kennt C++ z.B. noch Element- und Klassen-Funktionen, die immer dem Kontext einer Klasse zugeordnet sind; oder Static-Funktionen, die dem Kontext einer Übersetzungseinheit zugeordnet sind.

## 12.1 Einführung

Ein wichtiges Sprachelement von C++ sind Funktionen. Sie bieten die Möglichkeit, Codefragmente zusammenzufassen und parametrisiert von beliebiger Stelle aus zunutzen.

```
void f()
{
   cout << "Hallo Welt\n";
}

int main()
{
   f();
   f();
}
```

```
Ausgabe
Hallo Welt
Hallo Welt
```

Funktionen sind eins der zentralen Mittel für Wiederverwertung in C++. Außerdem sind sie ein leistungsfähiges Mittel um Code besser zu strukturieren. Immer wenn sie in C++ den gleichen oder ähnlichen Code zum zweiten Mal schreiben, sollten sie sich fragen, ob sie den Code nicht an einer Stelle zusammenführen und dann diesen einfach wiederverwenden können. Spätestens beim dritten Mal sollten sie sich das nicht nur fragen, sondern es auch machen! Funktionen sind ein Sprachmittel in C++ für diese Aufgabe.

### 12.1.1 Funktions-Aufruf

Eine Funktion wird mit ihrem Namen und den folgenden runden Klammern aufgerufen (benutzt). In den Klammern steht die sogenannte Argumentliste – eine durch Komma getrennte Liste von Argumenten, die auch leer sein kann. Die Argumente sind die Werte, die an die Funktion übergeben werden.

Syntax:
Funktionsaufruf :== Funktionsname(Argumentliste);

### 12.1.2 Funktions-Deklaration

Damit der Compiler in ISO C++ einen Funktionsaufruf compiliert, muss die Funktion deklariert werden – man nennt die Deklaration auch Bekanntmachung oder Prototyp. Durch die Deklaration wird dem Compiler die Funktion bekannt gemacht:

- Er kennt den Namen und weiß dass es sich dabei um eine Funktion handelt.
- Er kennt die Parameterliste, d.h. mit welchen Argumenten diese Funktion aufrufbar ist.
- Er kennt den Rückgabetyp.
- Und noch ein paar andere Dinge, die hier noch uninteressant sind, z.B. die Exception-Spezifikation.

Syntax (Funktions-Deklaration):
Rückgabetyp Funktionsname(Parameterliste);

```
// Beispiele fuer Funktions-Deklarationen

void f1();
int f2(int);
double f3(string s, char);
```

Die Deklaration besteht aus:

- Rückgabetyp – dies muss ein für den Compiler bekannter Typ sein, der im Normalfall kopierbar sein muss.
- Funktionsname – ein im Rahmen der C++ Namensregeln erlaubter und in seinem Kontext eindeutiger Name – über diesen Namen wird auf die Funktion später zugegriffen.
- Runde Klammer auf
- Parameterliste
  - Die Parameterliste darf leer sein.
  - Ansonsten ist sie eine durch Komma getrennte Liste von Parametern.
  - Ein Parameter ist ein Typ und ein optionaler Name
- Runde Klammer zu
- Abschliessendes Semikolon

```
int main()
{
   fct();      // Compiler-Fehler, da die Funktion fct nicht bekannt ist
}
```

```
void fct();    // Funktions-Deklaration

int main()
{
   fct();      // Compiliert, da die Funktion bekannt ist, und alles stimmt.
   fct(1);     // Compiler-Fehler, da die Argumente nicht zur Funktion passen.
}
```

**Hinweis** – eine Deklaration ist immer nur in ihrem Scope bekannt, d.h. auch Deklarationen unterliegen den normalen C++ Sichtbarkeitsregeln.

### 12.1.3 Funktions-Definition

Zusätzlich muss die Funktion natürlich noch irgendwo implementiert werden. Eine solche Implementierung wird Funktions-Definition oder auch Funktions-Implementierung genannt.

Syntax (Funktions-Definition)
Rückgabetyp Funktionsname(Parameterliste) { Funktionscode }

Der Funktionscode ist eine Folge von Deklarationen, Definitionen und Anweisungen, er kann aber auch leer sein.

```
// Beispiel fuer Funktions-Definitionen

void f1()
{
   cout << "Hallo Welt\n";
}

int f2(int x)
{
   int erg = 2*x-1;
   if (erg<0)
   {
      erg = 99;
   }
   return erg;
}

void f3(string s, char)
{
}
```

Hinweis – eine Funktions-Definition enthält auch immer implizit ihre Funktions-Deklaration, d.h. wenn die Funktion vor der Benutzung definiert wurde, ist eine explizite Deklaration nicht mehr notwendig.

```
void f()
{
}

// void f();    // Deklaration nicht notwendig

int main()
{
   f();         // Okay, da die Definition implizit die Deklaration enthaelt
}
```

## 12.2 Parameter und Argumente

In den meisten Fällen bekommen Funktionen zusätzliche Argumente übergeben, die – zur Laufzeit – in die Funktions-Verarbeitung integriert werden bzw. diese beeinflussen. Erst dadurch werden Funktionen wirklich leistungsfähig.

```
void print(int n)
{
   cout << n << ' ';
}

int main()
{
   for (int i=0; i<5; ++i)
   {
      print(i);
   }
```

```
}
```

**Ausgabe**
```
0 1 2 3 4
```

Namen:
- Auf der Seite der Funktion: Parameter – d.h. „n“ ist ein Parameter
- Auf der Seite des Funktions-Aufrufs: Argument – d.h. „i“ ist ein Argument

Parameter sind im Prinzip normale lokale Variablen der Funktion, nur das sie von außen an die Funktion beim Aufruf mitgegeben werden. Ansonsten gilt – wie für lokale Variablen:
- Sie sind nur innerhalb der Funktion sichtbar (zugreifbar).
- Mit Verlassen des Scopes (hier der Funktion) werden sie automatisch zerstört!
- Sie sind lokal zu diesem Funktionsaufruf, d.h. ein zweiter paralleller Funktionsaufruf enthält seinen eigenen Parameter (bzw. erzeugt seine eigene lokale Variable), die vollkommen unabhängig sind. Weiter unten findet sich dazu noch ein Beispiel.

Parameter werden an Funktionen entweder im sogenannten „call-by-value“ (cbv) oder „call-by-reference“ (cbr) Verfahren übergeben.

### 12.2.1 call-by-value

Im Normallfall werden Argumente in C++ per Wert übergeben. Das heißt, dass das an die Funktion eine **Kopie** des Arguments übergeben wird. Veränderungen des Parameters (d.h. der lokalen Kopie) betreffen das Argument (d.h. die Original-Variable) nicht!

```
void f(int arg)
{
   cout << "f1: " << arg << '\n';
   ++arg;
   cout << "f2: " << arg << '\n';
}

int main()
{
   int n = 4;
   cout << "m1: " << n << '\n';
   f(n);
   cout << "m2: " << n << '\n';       // n ist unveraendert 4
}
```

**Ausgabe**
```
m1: 4
f1: 4
f2: 5
m2: 4
```

Achtung – dies gilt für alle Typen, egal wie kompliziert sie auch aussehen mögen, z.B.:

```
void f(double);
void f(std::string);
void f(std::vector<int>);
void f(std::map<std::string, std::vector<int> >::const_iterator);
```

### 12.2.2 call-by-reference

Als Alternative zu „cbv“ gibt es „cbr“, bei dem **keine Kopie** des Arguments erzeugt wird, sondern statt dessen eine Referenz (ein Verweis) auf das Argument der Funktion übergeben wird. Der Parameter ist also keine lokale Kopie sondern nur eine lokale Referenz auf das Original Argument. Veränderungen am Parameter betreffen also nicht die lokale Referenz sondern das Original Argument – Referenzen selber können ja nicht verändert werden. Alle Veränderungen betreffen immer das von der Referenz referenzierte Objekt!

Um einen Parameter von einem cbv- zu einem cbr-Parameter zu wandeln, muss hinter dem Parameter-Typen und – wenn vorhanden – vor dem Parameter-Name das Kaufmanns-Und „&“ eingefügt werden.

```
void f(int& arg)                        // Hier steht jetzt zusaetzlich ein &
{
   cout << "f1: " << arg << '\n';       // <- hier wird ueberall ueber die
   ++arg;                               // <- Referenz 'arg' die Variable 'n'
   cout << "f2: " << arg << '\n';       // <- in main angesprochen.
}

int main()
{
   int n = 4;
   cout << "m1: " << n << '\n';
   f(n);
   cout << "m2: " << n << '\n';         // n ist 5
}
```

**Ausgabe**

```
m1: 4
f1: 4
f2: 5
m2: 5
```

Auch dies gilt natürlich für alle Typen, egal wie kompiliziert sie sind.

### 12.2.3 Wann was wie warum – und const-Referenzen

Im ersten Augenblick scheint die Sache klar zu sein:

- Soll die Original-Variable nicht verändert werden, nimm call-by-value.
- Soll die Original-Variable verändert werden, nimm call-by-reference.

Aber in C++ ist fast nichts so einfach, wie es zuerst ausschaut. Hier gilt noch zu bedenken, dass es in C++ Typen gibt:

- die nicht kopierbar sind (z.B. Streams), und
- bei denen eine Kopie teuer ist (speicher- und performance-mäßig, z.B. ein Vektor).

```
void f1(std::ostream);                  // nicht kopierbar
void f2(std::vector<std::string>);      // Kopie ist teuer
```

In beiden Fällen empfiehlt sich call-by-reference, auch wenn die Original-Variable nicht verändert werden soll. Damit fängt man sich aber zwei Probleme ein:

- Die Original-Variable könnte unbeabsichtigt doch verändert werden.
- Die Funktion ist für konstante Objekte nicht mehr nutzbar.

```
void f1(int arg)
{
   ++arg;              // Unbeabichtige Aenderung macht kein Problem
}

void f2(int& arg)
{
   ++arg;              // Unbeabichtige Aenderung IST EIN PROBLEM
}
```

```
void g(int&);

int main()
{
   const int i = 9;
   g(i);               // Compiler-Fehler, da i ueber g veraendert werden koennte!!
}
```

Beide Probleme können mit dem Modifier „const“ gelöst werden. Wird der Referenz-Parameter „const“ gemacht, so kann der Compiler Problem 1 finden, und Problem 2 gibt es nicht mehr.

```
void f(const int& arg)
{
   ++arg;              // Compiler-Fehler
}
```

```
void g(const int&);

int main()
{
   const int i = 9;
   g(i);               // Alles okay, da i ueber g nicht veraendert werden kann
}
```

Nun sind in der Praxis zwar erstaunlich viele Typen nicht kopierbar, aber sie sind trotzdem die Ausnahme. Viel wichtiger ist in der Praxis das Argument „Ressourcenverbrauch“ und hierbei vor allem das Thema Performance. Schon bei relativ einfachen und kleinen Objekten können die Performance-Kosten eines Kopiervorgangs höher sein als die Kosten durch indirekte Objektzugriffe.

### 12.2.4 Parameter-Übergabe-Regel

Für uns führt das zu der *“nicht-vollständigen“* Parameter-Übergabe-Regel:

- Sollen Original-Objekte in der Funktion verändert werden => call-by-reference
  - Es geht nicht anders – mit den anderen Aufruf-Arten können wir das Original-Objekt nicht ändern.
- Elementare Datentypen, Enums, Zeiger, usw... => call-by-value
  - Alle diese Typen (auch Basis-Daten-Typen) können von der Maschine sehr effizient gehandelt werden – daher ist die Übergabe per “cbv” effizient und unproblematisch.
- Alle anderen => call-by-const-reference
  - Alle anderen Typen können meist nicht effizient kopiert werden bzw. sind vielleicht gar nicht kopierbar – daher bietet sich die Referenz-Übergabe an. Da das Original-Objekt aber nicht geändert werden soll, nehmen wir natürlich Const-Referenzen.

In Wirklichkeit sieht die Regel etwas anders und viel komplizierter aus. Aber dazu fehlt uns

viel Wissen, z.B. über virtuelle Funktionen, Multithreading,…. Diese Features ändern im Einzelfall die Regel. Als Anfänger sollten Sie sich immer an diese Regel halten, dann machen Sie nie etwas falsch – höchstens vielleicht nicht ganz optimal. Und wenn Sie später als fortgeschrittener Programmierer bewußt und korrekt von dieser Regel abweichen, dann sollten Sie das immer kommentieren.

## 12.3 Rückgaben

Eine Funktion kann – aber muss nicht – ein Ergebnis zurückgeben – dies wird mit dem Rückgabe-Typ festgelegt.

- Soll eine Funktion nichts zurückgeben, so wird der Pseudotyp „void“ als Rückgabe-Typ in der Deklaration und Definition benutzt.
- Ansonsten kann fast jeder Typ als Rückgabe-Typ benutzt werden:
  - Er muss kopierbar sein – auch Rückgaben sind default-mässig „by-value“.
  - Natürlich sind auch Referenzen erlaubt – s.u.

Hat eine Funktion einen Ergebnis-Rückgabe-Typ (d.h. nicht „void“), so muss die Funktion mit einer „return“ Anweisung enden – Ausnahme ist nur die Funktion „main“, die ohne explizite „return“ Anweisung ein implizites „return 0;“ enthält.

Syntax: return ausdruck;

```
int f()
{
   return 2*4+8;
}
```

```
int f()
{
}    // Compiler-Fehler - es wird am Funktions-Ende keine Ergebnis zurueckgegeben.
```

Mit einer „return“ Anweisung wird die Funktion instantan beendet.

- Der Ausdruck in der „return“ Anweisung wird berechnet.
- Alle erzeugten lokalen Variablen und dieParameter werden in der umgekehrten Reihenfolge ihrer Konstruktion zerstört.
- Das Ausdrucks-Ergebnis wird zurückgegeben.

Eine Funktion kann beliebig viele Ausgänge haben, d.h. es können in einer Funktion beliebig viele return-Anweisungen stehen. Damit läßt sich verhindern, dass innerhalb der Funktion eine zu tiefe Schachtelung von Scopes entsteht.

```
int f(int arg)
{
   if (arg<0)
   {
      return -1;
   }
   if (arg==0)
   {
      return 0;
   }
   return 1;
}
```

Auch Funktionen ohne Rückgabe – d.h. mit „void“ als Rückgabe-Typ – können jederzeit beendet werden. In diesem Fall reicht eine return Anweisung mit leerem Ausdruck.

```
void f(int arg)
{
   if (arg<0)
   {
      return;
   }
   ...
}
```

### 12.3.1 Referenzen

Auch Referenzen können als Rückgabetyp eingesetzt werden.

```
int& fct(int& arg)
{
   arg *= 2;
   return arg;
}

int main()
{
   int i=9;
   cout << fct(i);               // -> 18
}
```

Das Beispiel sieht vielleicht etwas komisch aus, da der Parameter (in veränderter Form) zurückgegeben wird. Dies war aber notwendig, da es bei der Rückgabe von Referenzen eine Falle gibt: es dürfen nämlich **niemals** Referenzen auf lokale Variablen zurückgegeben werden. Und noch kennen wir keine Variablen, die länger als ein Funktionsaufruf leben.

```
// Fehlerhaftes Programm - UB - so niemals machen

int& f()
{
   int i = 42;
   return i;       // Achtung - gefaehrliches Laufzeitproblem - niemals machen! UB!
}

int main()
{
   cout << f();    // Worauf verweist die Referenz hier? UB!
}
```

Bedenken Sie, dass eine Referenz ein Verweis auf ein Objekt ist, in diesem Fall ist die Rückgabe daher der Verweis auf die lokale Variable „i“. Aber beim Verlassen der Funktion werden lokale Variablen automatisch zerstört, d.h. sie existieren nicht mehr. Die Referenz verweist also auf ein Objekt, das nicht mehr existiert. Der Zugriff erzeugt undefiniertes Verhalten.

**Hinweis** – das gefährliche an diesem Fehler ist, dass er in der Praxis meist lange Zeit nicht auffällt, da in vielen Fällen das passiert was man *unwissend* erwartet. Das lokale Objekt wird zwar zerstört, aber der Speicher existiert ja noch und meist ist der Zugriff vom Betriebssystem her noch erlaubt, und solange der Speicher nicht überschrieben ist, liefert der Zugriff das erwartete Ergebnis... Bis es dann doch mal schief geht – leider dann aber erst bei einer wichtigen Demo oder beim Kunden...

 

### 12.3.2 Nicht zwingende Nutzung

Wenn eine Funktion einen Wert zurückgibt, so kann dieser ignoriert werden.

```
int f()
{
   return 2;
}

int main()
{
   f();        // Rueckgabe wird nicht benutzt - das ist okay
}
```

Möchte man nicht, dass eine Rückgabe ignoriert wird, so kann man sie mit der Attribut-Spezifizierer-Sequens „nodiscard“ auszeichnen. Der Compiler muss dann eine Warnung erzeugen, wenn die Rückgabe nicht genutzt wird.

```
[[nodiscard]] int f()   // [[nodiscard]] => Rueckgabewert soll benutzt werden
{
   return 2;
}

int main()
{
   f();          // Rueckgabe wird nicht benutzt => Compiler-Warnung
   int x = f();  // Rueckgabe wird benutzt => alles okay
}
```

### 12.3.3 Weitere Syntax für Funktions-Rückgaben

In C++11 wurde eine neue Syntax für Funktions-Rückgaben eingeführt. Die Funktion beginnt hierbei mit dem Schlüsselwort „auto“ ohne Rückgabe-Typ, dann folgen Funktions-Name und Parameterliste, und danach dann der Pfeil-Operator und der (seit C++14 optionale) Rückgabe-Typ.

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

auto f() -> void                            // Neue C++11 Syntax
{
   cout << "void f()\n";
}

auto g(int arg) -> int                      // Neue C++11 Syntax
{
   cout << "int g(" << arg << ")\n";
   return 2*arg;
}

auto main() -> int                          // Auch fuer "main" geht die neue C++11 Syntax
{
   f();
   int n = g(3);
   cout << "n: " << n << '\n';
}
```

**Ausgabe**

```
void f()
int g(3)
n: 6
```

In C++14 wurde die Syntax noch weiter vereinfacht. Wenn dem Compiler die Funktions-Definition bekannt ist und alle Return-Anweisungen den gleichen Typ zurückgeben, dann

kann der Compiler den Rückgabe-Typ ja selber deduzieren. In diesem Fall darf man den Pfeil und den expliziten Rückgabe-Typ in der Funktions-Signatur weglassen:

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

auto f()                                        // Neue C++14 Syntax - ohne -> Typ
{
   return 2;
}

auto g(bool arg)                                // Neue C++14 Syntax - ohne -> Typ
{
   if (arg)
   {
      return 2.7;
   }
   return 3.14;
}

int main()
{
   cout << boolalpha;
   cout << "f(): " << f() << '\n';
   cout << "g(true):  " << g(true) << '\n';
   cout << "g(false): " << g(false) << '\n';
}
```

**Ausgabe**

```
f(): 2
g(true):  2.7
g(false): 3.14
```

### 12.3.4 Rückgabe neuer Objekte

In der Praxis ist es nicht selten, dass Sie neue Objekte aus einer Funktion zurückgeben wollen, die Sie vorher gar nicht zur Verfügung haben. Im folgenden das ziemlich sinnlose Beispiel, dass einen String zurückgibt, der

- genauso lang ist wie der übergebene String und nur aus dem ersten Buchstaben des übergebene Strings besteht, bzw.
- im Falle eines übergebenen Leerstrings auch leer ist.

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

string createNewString(const string& in)
{
   if (in.empty())
   {
      string res;                // (*)
      return res;                // Okay, man koennte auch "in" zurueckgeben...
   }

   string::size_type len = in.length();
   char first = in[0];
   string res(len, first);       // (**)
   return res;
}

int main()
{
   cout << "<>    => " << createNewString("") << '\n';
   cout << "<a>   => " << createNewString("a") << '\n';
   cout << "<xyz> => " << createNewString("xyz") << '\n';
}
```

```
Ausgabe
<>    =>
<a>   => a
<xyz> => xxx
```

Interessant sind hier besonders die Zeilen (*) und (**). In beiden Zeilen wird das jeweilige Result-Objekt „res“ konstruiert, das dann in der Zeile danach mit „return“ zurückgegeben wird. Wir ignorieren hierbei mal, dass man in (*) auch direkt „in“ hätte zurückgeben können, und diese Optimierung auch für Strings der Länge „1“ anwendbar wäre.

Man kann das Rückgabe-Objekt auch direkt in der Return-Anweisung konstruieren, indem man die funktionale Konvertierung aus Kapitel 11.4.2 anwendet „typ(initialwerte)“. Genau genommen wird hier durch die Konvertierung ein temporäres Objekt erzeugt.

```
string createNewString(const string& in)
{
   if (in.empty())
   {
      return string();                    // <= Objekt direkt beim "return" erzeugen
   }

   string::size type len = in.length();
   char first = in[0];
   return string(len, first);             // <= Objekt direkt beim "return" erzeugen
}
```

Seit C++11 kann man hier auch eine implizite Konvertierung (siehe Kapitel 11.4.1) mit den geschweiften Klammern nutzen, die wir noch näher kennen lernen werden. Aber Achtung, da lauert die gleiche Falle wie in der normalen Initialisierung mit geschweiften Klammen, dass die Sequenz-Konstruktion vorgeht.

```
string createNewString(const string& in)
{
   if (in.empty())
   {
      return {};                     // <= geschweifte Klammern erzeugen hier leeren String
   }

   string::size_type len = in.length();
   char first = in[0];
   return string(len, first);        // <= geschweifte Klammern gehen hier leider nicht
}                                    //    Erzeugen hier eine Sequenzkonstruktion
```

Besonders bei komplexen Typ-Namen – wie z.B. Containern – ist die Möglichkeit mit den geschweiften Klammern sehr hilfreich. Aber Achtung – sie läßt sich nicht mit der C++14 Rückgabe-Typ Deduktion aus Kapitel 12.3.3 kombinieren. Denn die Typ-Deduktion benötigt den Typ der Return-Anweisung, und die geschweiften Klammern den Rückgabe-Typ zur Konvertierung – sie bauen also gegenseitig aufeinander auf, d.h. muss mindestens eine Information vorhanden sein.

Hinweis – kommen Sie nicht auf die Idee, bei der Rückgabe neuer Objekte aus Performance-Gründen statt der Kopie eine Referenz zurückzugeben. Neue Objekte sind eigentlich immer lokale Objekte, und die Rückgabe einer Referenz auf lokale Objekte erzeugt UB – siehe oben. Zum Glück ist ist die Rückgabe neuer Objekte als Kopie meistens kein Problem, da der Compiler diese Kopie häufig wegoptimieren kann, bzw. seit C++17 zum Teil sogar muss (Stichwort „copy elision“), bzw. die Objekte „moven“ kann. Die genauen

Details sprengen mal wieder den Rahmen der Vorlesung – die Begriffe, die Ihnen hier bei der Suche im Internet weiterhelfen sind „copy elision“, „Move-Semantik“, „RVO“ (Return Value Optimization) und „NRVO“ (Named Return Value Optimization).

## 12.4 Konvertierungen

Wie schon in Kapitel 11.4.1 erwähnt, kann es auch bei Funktions-Aufrufen und Funktions-Rückgaben zu impliziten Konvertierungen kommen. Im Prinzip ist dies nichts anderes als die normalen Konvertierungen innerhalb von Ausdrücken – nur bei Funktionen sieht man sie oft expliziter.

Wenn eine Funktion nicht die genau passenden Parameter-Typen erwartet, und die Sprache entsprechende Konvertierungen erlaubt – dann konvertiert der Compiler die Argumente implizit.

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

void f(int n)
{
   cout << "f(" << n << ")\n";
}

void g(double d)
{
   cout << "g(" << d << ")\n";
}

int main()
{
   f(2);                          // Aufruf von "f" mit passendem "int"
   f(3.14);                       // Aufruf von "f" mit "double" => konvertiert zu "int"

   g(8);                          // Aufruf von "g" mit "int" => konvertiert zu "double"
   g(9.23);                       // Aufruf von "g" mit passendem "double"
}
```

**Ausgabe**

```
f(2)
f(3)
g(8)
g(9.23)
```

Während diese Konvertierungen („int“ => „double“ bzw. „double“ => „int“) relativ harmlos aussehen – aber trotzdem ab und zu für Fehler sorgen können – gibt es auch Konvertierungen, die weitaus „größer“ und in der Praxis oft sehr hilfreich sind, z.B. für Strings.

Sie erinnern sich hoffentlich daran, dass Zeichenketten-Konstanten keine Strings sind. Trotzdem kann man Funktionen schreiben, die Strings erwarten und sich mit Zeichenketten-Konstanten aufrufen lassen – implizite Typ-Konvertierungen sei Dank.

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

void f(const string& s)
{
   cout << "f(" << s << ")\n";
```

```
}

int main()
{
   string str("std::string");
   f(str);                          // Aufruf von "f" mit "string"
   f("Kein std::string");           // Aufruf von "f" ohne "string" => Typ-Konvertierung
}
```

**Ausgabe**
```
f(std::string)
f(Kein std::string)
```

Dies ist ein schönes Beispiel für eine implizite Konvertierung, die in der Praxis sehr hilfreich ist – denn keiner möchte auf diesen Aufruf-Komfort verzichten. Und eine, die wir auch schon mehrfach in den Beispielen benutzt haben – ohne dass irgendjemand stutzig geworden ist oder sich gewundert hat. Nur umgekehrt – würde das nicht funktionieren – dann hätten wir uns gewundert und gemeckert.

Diese impliziten Typ-Umwandlungen stehen natürlich nicht nur für die Funktions-Argumente, sondern auch für die Funktions-Rückgabe zur Verfügung.

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

double h()
{
   return 2.7;
}

int main()
{
   double d = h();                  // Rueckgabe ist "double" - alles okay
   cout << "d: " << d << '\n';

   int n = h();                     // Rueckgabe ist "double", kein "int" => Konvertierung
   cout << "n: " << n << '\n';
}
```

**Ausgabe**
```
d: 2.7
n: 2
```

Der Compiler darf pro impliziter Konvertierungen immer beliebig viele von der Sprache definierte Typ-Umwandlungen, und max. eine benutzerdefinierte Typ-Umwandlung vornehmen. Benutzerdefinierte Typ-Umwandlungen werden wir in Form von Konvertierungs-Konstruktoren kennenlernen. Außerdem gibt es noch benutzerdefinierte Konvertierungs-Operatoren, die ein Teil des Themas Operator-Überladung sind, die in der Vorlesung aber nicht besprochen werden.

Natürlich ist man nicht auf die implizite Typ-Konvertierung angewiesen. Wenn man möchte – weil sie z.B. nicht gut lesbar oder verwirrend ist – dann kann man sie auch explizit machen. Dazu nutzt man die Cast-Operatoren aus Kapitel 11.4.2, häufig in Form von „static_cast“ oder dem funktionalen C++ Stil.

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

void f(int n)
{
```

```
   cout << "f(" << n << ")\n";
}

void g(const string& s)
{
   cout << "g(" << s << ")\n";
}

int main()
{
   f(static_cast<int>(3.14));       // Explizite Konvertierung mit "static_cast"
   f(string("String"));             // Explizite Konvertierung im funktionalen C++ Stil
   f("Literal"s);                   // Hier noch besser der Literal Postfix fuer Strings
}
```

**Ausgabe**

```
f(3)
g(String)
g(Literal)
```

## 12.5 Default-Argumente

In der Praxis wäre es oft angenehm, bei einem Funktionsaufruf nicht alle Parameter angeben zu müssen, z.B. wenn:

- ein Parameter fast immer den gleichen Wert hat,
- ein Parameter bzw. seine Auswirkung in bestehenden Kontext nicht interessiert, oder
- oder Benutzer einfach keinen guten Defaultwert kennt.

In allen diesen Fällen wäre es schön, wenn der Entwickler der Funktion die Parameter bzw. zumindest einen Teil davon mit sinnvollen Defaultwerten belegt hätte. Beispiele hierfür sind z.B. die „ignore“ Element-Funktion der Eingabe-Streams, oder die Element-Funktion „substr“ von „string“.

In C++ können nämlich Default-Argumente vorgegeben werden, die automatisch vom Compiler bei einem Aufruf ohne diese Parameter benutzt werden. Dazu werden die Default-Argumente einfach in der Funktions-Parameterliste den Parametern zugewiesen.

```
void f(int=1);

f();       // => f(1)
f(4);      // => f(4)
```

- Es können beliebig viele Default-Argumente benutzt werden - sie müssen aber ohne Lücke immer die letzten Parameter der Signatur abdecken, da sonst Mehrdeutigkeiten auftreten könnten.
- Default-Argumente dürfen auch in Verbindung mit Parameternamen auftauchen.

```
void f(int i=4, char c='A', long l=17);

f();            // => f(4, 'A', 17)
f(0);           // => f(0, 'A', 17)
f(2, 'B');      // => f(2, 'B', 17)
f(5, 'M', 12);  // => f(5, 'M', 12)
```

```
// Funktionen mit fehlerhaften Default-Argumenten.
// Die Default-Argumente decken die letzten Parameter nicht buendig ab

void f1(int=4, char, long=17);       // Compiler-Fehler
void f2(int, int=9, int);            // Compiler-Fehler
```

**Praxis** – der Compiler muss die Default-Argumente kennen, wenn er sie beim Aufruf einsetzen soll. Selbst wenn wir noch keine Aufteilung unserer Programme auf mehrere Quelltexte kennen – merken Sie sich hier schon mal, dass es sich daher empfiehlt, die Default-Argumente in den Funktions-Deklarationen in den Headerdateien aufzuführen.

## 12.6 Überladen

In C++ gehört die Parameterliste einer Funktion (die sogenannte Signatur) zum Funktions-Namen, d. h. unterschiedliche Funktionen können den gleichen Namen haben, solange sie sich durch ihre Signatur unterscheiden. Dies nennt man Überladen oder vollständiger „Funktions-Überladung“.

```
int f();
int f(bool);
int f(int);
int f(char);
int f(signed char);
int f(unsigned char);
int f(long, int);
int f(string&, int);
int f(string&);
int f(const string&);
```

**Achtung** – der Rückgabetyp zählt nicht zur Signatur bzw. zum Funktions-Namen und trägt daher nicht zur Unterscheidung bei. Man kann Funktionen also **nicht** nur mit dem Rückgabe-Typ überladen.

Beim Aufruf einer Funktion entscheidet der Compiler anhand der Argumente, welche Funktion er nimmt.

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

void f(int)                                     // Funktion f mit int
{
   cout << "int\n";
}

void f(char)                                    // Funktion f mit char
{
   cout << "char\n";
}

void f(int, char)                               // Funktion f mit int und char
{
   cout << "int, char\n";
}

int main()
{
   f(4);                    // Ausgabe:  int
   f('C');                  // "         char
   f(2, 'A');               // "         int, char
}
```

**Hinweis** – auch z.B. *Referenz-auf-Typ* und Const-*Referenz-auf -Typ* sind beim Überladen unterschiedliche Datentypen. Solange der Compiler beim Aufruf anhand der Argument-Typen die Funktionen unterscheiden kann, ist alles in Ordnung – vergleiche auch die Diskussion in Kapitel 12.6.2.

Überladen sieht zuerst nur nach einem kleinen netten Feature in C++ aus, ist aber an vielen Stellen sehr wichtig für den Aufbau, die Flexibilität und die Erweiterbarkeit der Sprache. Ein Beispiel für das Überladen als grundlegendes Feature für die Erweiterbarkeit von C++ werden wir z.B. in Form der typsicheren und benutzer-definierten Ausgabe auf Streams sehen. Aber z.B. auch die freien Funktionen „begin“ und „end“ bei der Nutzung von Iteratoren sind Beispiele für die Flexibilität und Erweiterbarkeit die Überladen in die Sprache C++ bringt. Ein weiteres werden wir im folgenden Kapitel 12.6.1 kennen lernen.

### 12.6.1 Flexibilität

Ein konkretes Beispiel für die Flexibiltät von C++ durch u.a. das Feature Überladen ist die bekannte mathematische Sinus-Funktion. Diese Funktion steht in C++ mit der Funktion „sin“ aus dem Header „cmath“ zur Verfügung.

Nehmen wir also an, Sie hätten ein großes mathematisches Problem zu lösen, in dessen Kontext u.a. die Sinus-Funktion vorkommt. Da Sie noch nicht abschätzen können, welche exakten Performance- und Genauigkeits-Anforderungen an Ihren Lösungs-Algorithmus gestellt werden, gehen Sie auf Nummer Sicher und verwenden für Ihren mathematischen Basis-Typ nicht „float“, „double“ oder „long double“, sondern einen Typdef „myfloat“.

```
using myfloat = double;
```

Nun haben Sie die Möglicheit, den zugrunde liegenden Typ leicht zu ändern und sind offen für jede kommende Anforderung – vergleiche auch Kapitel 11.1.2.4.

Nun kommen Sie beim Implementieren an die Stelle, wo Sie den Sinus benötigen:

```
// Problem ohne Ueberladen:

myfloat calc_it(myfloat in)
{
   myfloat res = sin(in);              // <= Oh je, welches Sinus: "sinf", "sin", "sinl"?
   res *= 2.;
   res += 3.14;
   return res;
}
```

In einer Sprache ohne Überladen (z.B. beim Vorgänger „C“) würde Ihnen jetzt die gesamte Anpassbarkeit von Typ-Aliasen nichts bringen – C90 kennt die Sinus-Funktion nur für Double-Werte. In C99 wurden dann zusätzlich noch „sinf“ für Float- und „sinl“ für Long-Double-Zahlen eingeführt. Aber trotzdem müssen Sie nun entscheiden, welche Funktion Sie aufrufen und jede Änderung Ihres Mathe-Typs müßte zu einer Änderung hier beim Aufruf führen – eine Wartungshölle.

Wieviel einfach dagegen in C++ mit Überladen. Natürlich enthält C++ die Sinus-Funktion überladen für „float“, „double“, „long double“ und auch noch andere Typen wie z.B. komplexe Zahlen oder Werte-Vektoren. Und der Compiler erkennt automatisch am Argument-Typ von „in“ welche Funktion er nehmen muss – da der Name immer gleich ist.

 

```
// Kein Problem in C++ dank Ueberladens

myfloat calc_it(myfloat in)
{
   myfloat res = sin(in);              // <= Compiler nimmt immer die richtige Funktion
   res *= 2.;
   res += 3.14;
   return res;
}
```

Sie müssen auf „float“ umschwenken – vielleicht um Speicherplatz zu sparen: kein Problem – Typedef ändern, neu compilieren, fertig. Oder Sie benötigen einen „long double“, weil die Genauigkeit über alles geht: kein Problem – Typedef ändern, neu compilieren, fertig. Das ist einfacher flexibler und wartungsfreundlicher Code – so sollte es sein.

### 12.6.2 Konvertierungs-Hierarchien und Mehrdeutigkeiten

Leider gibt es im Detail dann doch ein Problem mit Überladen: was ist wenn es keine exakt passende Funktion gibt, aber mehrere prinzipiell mögliche Funktionen?

```
void f(short);
void f(long);

int main()
{
   f(4);      // "4" ist ein "int"
}             // -> Welche Funktion wird aufgerufen? "short" oder "long" oder was passiert?
              // Hier Compiler-Fehler
```

Diese Frage läßt sich nicht so einfach allgemein beantworten – in diesem konkreten Beispiel („int=>long“ oder „int=>short“) bekommt man einen Compiler-Fehler, da hier der Funktions-Aufruf für den Compiler mehrdeutig ist. Die Sprache erlaubt implizite Konvertierungen vom „int“ zum „long“ und auch zum „short“. Beide Konvertierungen sind integrale Konvertierungen, und aus Sicht der Sprache absolut gleichwertig – darum erzeugt dies einen Compiler-Fehler.

**Achtung** – lassen Sie sich nicht täuschen. Ein „short“ könnte kleiner als ein „int“ sein, während der „long“ mindestens so groß wie ein „int“ ist. Die Wandlung von einem „int“ zu einem „long“ sieht hier also viel sinnvoller aus, da in diese Richtung niemals ein Datenverlust stattfindet. Leider interessieren sich die Konvertierungs-Regeln nicht für solche Überlegungen – ihre Wurzeln sind mittlerweile weit über 40 Jahre alt und damals waren andere Anforderungen interessant. Entgegen der Erwartung des Anfängers gilt hier: beide Konvertierungen sind möglich und auch noch gleichwertig => Compiler-Fehler.

Nun gilt dies aber nicht für alle Wandlungen – sie sind **nicht** alle gleichwertig. Ganz im Gegenteil sind alle Konvertierungen in C++ sind Teil einer Konvertierungs-Hierarchie. Die genauen Regeln sind recht kompliziert und sprengen leider bei weitem den Umfang der Vorlesung.

Zurück zum eigentlichen Thema: alle Konvertierungen sind Teil einer Konvertierungs-Hierarchie, die für uns zurzeit ungefähr so aussieht, wobei die gleichzeitige Nutzung von exakter Übereinstimmung (Kopie) nicht gleichzeitig mit Referenzen genutzt werden darf:

1. Exakte Übereinstimmung
   - D.h. keine Konvertierung notwendig
2. Triviale Wandlungen ohne semantische Änderungen
   - Typ => Typ&
   - Const Typ => const Typ&
3. Triviale Wandlungen mit semantische Änderungen
   - Typ => const Typ&
4. Numerische Promotionen (Integrale und Fließkomma Promotionen)
   - char => int
   - short => int
   - float => double
   - und viele weitere…
5. Numerische Konvertierungen (Integrale und Fließkomma Konvertierungen)
   - int => double
   - int => long
   - long long => short
   - und viele weitere…
6. Weitere Sprach-Konvertierungen
   - Unscoped Enums zu Integer

Ist eine Konvertierung in dieser Hierarchie eher vorhanden, so greift sie – ohne dass es zu einer Mehrdeutigkeit kommt. Konvertierungen auf der gleichen Hierarchie-Ebene führen zu Mehrdeutigkeiten und damit zu Compiler-Fehlern.

Aus dieser Hierarchie ergibt sich u.a., daß sich z.B. eine Referenz und eine Const-Referenz überladen lassen. Ein Non-Const Objekt bindet dabei eher an die normale Referenz (Stufe 2) als die Const-Referenz (Stufe 3), während ein Const-Objekt nur an die Const-Referenz binden kann (siehe Kapitel 11.2.1), was hier Stufe 2 entspricht:

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

void f(string& s)                                   // Funktion f mit String Referenz
{
   cout << "Non-Const String Referenz\n";
}

void f(const string& s)                              // Funktion f mit Const String Referenz
{
   cout << "Const String Referenz\n";
}

int main()
{
   string s("Non-Const");
   const string cs("Const");

   f(s);      // Nimmt f(string&), da non-const Objekte eher an non-const Referenzen binden
   f(cs);     // Nimmt f(const string&), da const Objekte nur an const Referenzen binden
}
```

**Ausgabe**

```
Non-Const String Referenz
Const String Referenz
```

Was aber nun, wenn eine Mehrdeutigkeit existiert – d.h. ein Compiler-Fehler. Wie bringt man solchen Code zum Laufen? Nun, hierbei gibt es zwei Möglichkeiten:

- Sie erzeugen die fehlende passende Funktion
- Oder Sie schreiben eine explizite Typ-Konvertierung

**1. Passende Funktion erzeugen:**

```
void f(short);
void f(int);       // Neue Funktion "f" fuer "int"
void f(long);

f(4);              // Jetzt okay -> f(int)
```

**2. Explizite Typkonvertierung:**

```
void f(short);
void f(long);

f(static_cast<long>(4));      // Explizite Konvertierung -> okay -> f(long)
```

## 12.7 Parameter und lokale Variablen

Parameter, die keine Referenzen sind, und lokale Variablen sind in C++ wirklich funktionslokal, d.h. **jeder** Funktionsaufruf bekommt seinen eigenen Satz an Parametern und lokalen Variablen.

```
void f(int arg)
{
   int loc = arg + 1;

   cout << "f(" << arg << ")\n";
   cout << "- loc: " << loc << '\n';

   arg += 10;
   loc += 20;

   cout << "- arg: " << arg << '\n';
   cout << "- loc: " << loc << '\n';

   if (arg==10)
   {
      cout << "  arg==10 => return\n";
      return;
   }


   f(0);

   cout << "- arg: " << arg << '\n';
   cout << "- loc: " << loc << '\n';
   cout << "  return\n";
}

int main()
{
   f(5);
}
```

**Ausgabe**

```
f(5)
- loc: 6
- arg: 15
- loc: 26
f(0)
- loc: 1
```

```
- arg: 10
- loc: 21
  arg==10 => return
- arg: 15
- loc: 26
  return
```

Hier kann man sehr schön sehen, dass der erneute Aufruf der Funktion „f“ seine eigenen Parameter und lokalen Variablen bekommt, und deren Veränderung keine Veränderung an den Parametern und lokalen Variablen der ersten Aufrufs vornimmt.

Diesen Effekt – dass eine Funktion zur gleichen Zeit mehrfach aufgerufen wird – bekommt man z.B. bei Rekursionen oder Multi-Threading.

## 12.8 Rekursion

Wenn eine Funktion im gleichen Thread mehrfach **ineinander** (d.h. nicht nacheinander, sondern gleichzeitig parallel) aufgerufen wird, nennt man das „Rekursion“ bzw. „rekursive Programmierung“.

Eine Funktion muss sich dabei nicht zwangsläufig selber aufrufen, sondern dies kann auch über mehrere Zwischenstationen passieren, z.B. f -> g -> h -> i -> j -> f. Solche Fälle sind häufig gar nicht mehr sofort zu erkennen, von daher passiert dies in der Praxis häufiger, als man vielleicht im ersten Augenblick denkt.

Es gibt aber auch viele Probleme, die sich rekursiv viel viel leichter programmieren lassen als ohne Rekursion. Hier ein Beispiel, das man in der Praxis sicher nicht rekursiv lösen würde – die Summe aller Zahlen von 1 bis n.

```
int sum(int arg)
{
   if (arg<=1)                // Operator <= statt == um die Funktion bei fehlerhaften
   {                          // Argumenten (arg<1) sauber zu beenden.
      return 1;
   }
   int erg = arg + sum(arg-1);
   return erg;
}

int main()
{
   cout << sum(5) << '\n';
}
```

```
Ausgabe
15
```

Hierbei wird quasi direkt die mathematische Definition umgesetzt:
sum(1) := 1
sum(n) := n + sum(n-1)

Natürlich läßt sich die Summe der Zahlen von 1 bis n direkt über die Formel „n*(n+1)/2“ berechnen – was hier auch viel sinnvoller wäre – aber es geht eben auch rekursiv.

**Hinweis** – es läßt sich übrigens beweisen, dass jedes Problem was iterativ gelöst werden kann (d.h. mit einer Schleife) sich auch rekursiv lösen läßt, und umgekehrt.

**Praxis** – in den aller-meisten Fällen sind die iterativen Lösungen schneller und benötigen weniger Speicher – sie sind daher vorzuziehen. Ein Schleifen-Durchlauf ist einfach schnell und effizient, während ein Funktions-Aufruf doch relativ *teuer* ist (bezogen auf die Performance). In manchen Fällen ist die rekursive Lösung aber ein 5-Zeiler, während die iterative Lösung harte Arbeit sein kann und hinterher aus vielen Zeilen schwer lesbarem Quelltext besteht.

## 12.9 Generische Funktionen

Stellen Sie sich eine einfache Minimums-Funktion „mini“ vor, die die kleinste zweier Int-Zahlen zurückgibt.

```
int mini(int a1, int a2)
{
   return a1<a2 ? a1 : a2;
}
```

Diese Funktion ist sehr hilfreich, hat aber einen entscheidenden Nachteil. Sie funktioniert nur mit Int-Werten. Benötigen Sie zusätzlich die Minimums-Funktion für zum Beispiel „double“, so müssen Sie eine weitere Funktion implementieren.

```
double mini(double a1, double a2)
{
   return a1<a2 ? a1 : a2;
}
```

Und natürlich werden Sie auf Dauer die Minimums-Funktion auch noch für „short“, „unsigned int“, „std::string“ und viele andere Typen benötigen.

### 12.9.1 Templates

Mit Templates können Sie in C++ generischen Code programmieren. Daher Code, bei dem die Typen variant sein können. Eingeleitet werden Templates mit dem Schlüsselwort „template“, und danach folgen in spitzen Klammern die Template-Parameter. Dabei werden Typ-Parameter (es gibt auch noch andere, aber dazu später mehr) entweder mit dem Schlüsselwort „class“ oder „typename“ eingeleitet. Beide Varianten sind absolut identisch – es gibt zwei Varianten aus historischen Gründen. Nach „class“ oder „typename“ folgt dann der Name des varianten Typs. Diese Typen stehen danach in der folgenden Funktions-Deklaration oder –Definition als gewöhnliche Typen zur Verfügung.

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

// Funktions-Deklarationen
template<class T> void f(T);
template<class A, typename xyz> void fct(const A&, xyz);

// Funktions-Definition
```

```
template<class T> void f(T t)
{
   cout << "f(" << t << ")\n";
}

// Funktions-Definition
template<class A, typename xy> void fct(const A& a, xy z)
{
   cout << "fct(" << a << ", " << z << ")\n";
}
```

Diese Funktions-Templates können dann wie normale Funktionen genutzt werden. Der Compiler deduziert die Typen der Parameter anhand der Typen der Argumente (mit den gleichen Regeln wie „auto“) und generiert eine entsprechende Funktion mit den konkreten Typen.

```
int main()
{
   f(42);
   f(3.14);
   f("Hallo");

   fct("Die halbe Wahrheit ist"s, 21);
   fct(6, 2.71);
}
```

**Ausgabe**
```
f(42)
f(3.14)
f(Hallo)
fct(Die halbe Wahrheit ist, 21)
fct(6, 2.71)
```

Die varianten Typ-Parameter können dabei auch indirekt in den Funktions-Parametern vorkommen, zum Beispiel als „vector<T>“.

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

template<class T> void push_element(vector<T>& v)
{
   T t = T();
   v.push_back(t);
}

int main()
{
   vector v { 1, 2, 3, 4 };
   push_element(v);
   for (int x : v)
   {
      cout << x << ' ';
   }
}
```

**Ausgabe**
```
1 2 3 4 0
```

Alle Template-Deduktionen, -Auflösungen und -Generierungen werden dabei vom Compiler vorgenommen und geschehen daher zur Compile-Zeit. Templates haben keinen Laufzeit-Overhead, daher sie beeinflussen die Performance nie negativ. Ganz im Gegenteil kann der Compiler bei Templates häufig sehr gut optimieren, da er zur Compile-Zeit alle Informationen

aufgelöst vorliegen hat.

Seit C++20 gibt es eine Kurzschreibweise für Funktions-Templates. Man benutzt einfach „auto“ Parameter in der Funktions-Signatur. Aber Achtung – im Hintergrund sind das immer noch Funktions-Templates (wie man z.B. auf CppInsights (https://cppinsights.io/) verifzieren kann), und er werden weiterhin vom Compiler für alle Typen eigene Implementierungen erzeugt.

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

// Weiterhin Funktions-Template
void f(auto t)
{
   cout << "f(" << t << ")\n";
}

int main()
{
   f(42);
   f(3.14);
   f("Hallo");
}
```

**Ausgabe**
```
f(42)
f(3.14)
f(Hallo)
```

Man kann Funktions-Templates mit normalen Funktionen und mit anderen Funktions-Templates überladen (egal, ob in alter Template-Schreibweise, oder in neuer C++20 Kurzschreibweise). Für den Aufruf wird zuerst eine normale Funktion mit exaktem Match genommen (triviale Wandlungen sind erlaubt), und danach ein Funktions-Template genommen. Matchen mehrere Funktions-Templates, so wird das Passenste genommen.

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

template<class T> void f(T t)
{
   cout << "f(" << t << ")\n";
}

template<class T> void f(const vector<T>& t)
{
   cout << "f(const vector<T>&)\n";
}

void f(int t)
{
   cout << "f(int " << t << ")\n";
}

void f(float t)
{
   cout << "f(float " << t << ")\n";
}

int main()
{
   vector<int> v;

   f(42);          // Exakter Match => int
```

```
    f(42L);          // Kein exakter Match => T
    f(42LL);         // Kein exakter Match => T
    f(3.14F);        // Exakter Match => float
    f(3.14);         // Kein exakter Match => T
    f("Hallo");      // Kein exakter Match => T
    f(v);            // Kein exakter Match, aber vector<T> ist besser als T
}
```

**Ausgabe**

```
f(int 42)
f(42)
f(42)
f(float 3.14)
f(3.14)
f(Hallo)
f(const vector<T>&)
```

Funktions-Templates kann man sowohl in der normalen Form als auch in der Kurzform mit Concepts kombinieren. In der normalen Form wird das Schlüsselwort „class“ oder „typename“ durch das Concept ersetzt. In der Kurzform wird, wie bei den Auto-Variablen, das Concept vor das Schlüsselwort „auto“ eingefügt.

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

template<class T> void f(T t)                // Allgemein T
{
   cout << "f(T " << t << ")\n";
}

template<integral T> void f(T t)             // Integrale-Typen
{
   cout << "f(integral " << t << ")\n";
}

void f(floating_point auto t)                // Fliesskomma-Typen
{
   cout << "f(floating_point " << t << ")\n";
}

void f(int t)                                // Exakt int
{
   cout << "f(int " << t << ")\n";
}

int main()
{
   f(42);          // Exakter Match => int
   f(42L);         // Integraler Typ => integral
   f(3.14);        // Fliesskomma Typ => floating_point
   f("Hallo");     // Kein Match => T
}
```

**Ausgabe**

```
f(int 42)
f(integral 42)
f(floating_point 3.14)
f(T Hallo)
```

## 12.10 Constexpr Funktionen

Am Anfang des Buchs haben wir gelernt, wie wir mit „const“ oder „constexpr“ Variablen zu Konstanten transformieren können. Der Unterschied zwischen „const“ und „constexpr“ war,

dass „const“ Variablen sowohl Compile- als auch Lauf-Zeit Konstanten sein konnten, während „constexpr“ Variablen immer zur Compile-Zeit bestimmbar sein mussten.

```
const int x = 4;                            // Compile-Zeit Konstante
const int y = leseZahlVomBenutzerEin();     // Lauf-Zeit Konstante
constexpr int z = 6;                        // MUSS eine Compile-Zeit Konstante sein
```

Natürlich wäre es schön, auch Compile-Zeit Konstanten mit einer Funktion initialisieren zu können, da sich die Initial-Werte nicht immer so einfach schreiben lassen.

```
constexpr int xyz = berechneWert();
```

Ein Funktions-Aufruf geschieht aber eigentlich zur Lauf-Zeit, während die Compile-Zeit Konstante schon zur Compile-Zeit initialisiert werden muss. Darum führt folgender Code zu einem Compile-Fehler.

```
int returnSix()
{
   return 6;
}

constexpr int six = returnSix();      // Compile-Fehler
```

Was wir hierfür benötigen, ist eine Funktion, die man schon zur Compile-Zeit aufrufen und ablaufen lassen kann. Genau das sind „constexpr“ Funktionen in C++. Funktionen mit einem vorgestellten „constexpr“ können sowohl zur Laufzeit, als auch zur Compile-Zeit ausgeführt werden.

```
constexpr int returnSix()             // Funktion ist nun "constexpr"
{
   return 6;
}

constexpr int six = returnSix();      // Okay, Compile-Zeit Konstante
```

Hierbei muss natürlich die gesamte Funktion prinzipiell zur Compile-Zeit ausführbar sein, und alle genutzten Variablen müssen zur Compile-Zeit schon feststehen. Werden zum Beispiel andere Funktionen aufgerufen, so müssen diese natürlich auch wieder „constexpr“ sein. Alle Funktionen, die zum Beispiel von Benutzer-Eingaben abhängig sind, können daher nie zur Compile-Zeit ausgeführt werden. Aber auch viele Funktionen, die von Laufzeit-Effekten abhängen, können nicht zur Compile-Zeit ausgeführt werden, wie zum Beispiel String Operationen.

Der große Vorteil von constexpr Funktionen ist aber nicht, dass man mit ihnen elegant Konstanten initialisieren kann, sondern dass sie zur Performance beitragen. Alle Berechnungen, die schon zur Compile-Zeit passieren, benötigen keine Laufzeit mehr und beschleunigen damit den Programm-Ablauf.

Mit C++20 sind zusätzlich „consteval“ Funktionen eingeführt worden, die nur zur Compile-Zeit aufgerufen werden dürfen.

 