C++ FOR (Monday Morning)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Falls die Übungsaufgabe nicht wegen eines verkürzten Vormittagsteils entfällt, erfolgt die Besprechung der Musterlösung(en) im direkten Anschluss an die Mittagspause.

Auffrischung einiger wichtiger Grundlagen

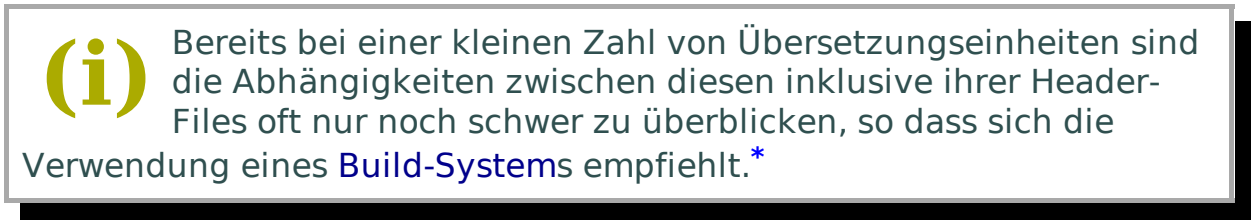
- Getrennte Kompilierung
- Definition/Reference-Modell
- Schreibschutz durch den Compiler

Getrennte Kompilierung

Grundlagen der getrennten Kompilierung

Ein C++-Programm wird üblicherweise in eine mehr oder weniger große Zahl von Übersetzungseinheiten aufgeteilt.

- Diese stehen in Implementierungsdateien, deren Dateinamens-Suffix meist `.cpp` ist.
- Sind ein und dieselben Informationen in mehr als einer Übersetzungseinheit notwendig, gehören diese in Header-Files, deren Dateinamens-Suffix meist `.h` ist (seltener: `.hpp`).



*: Unter den heute verwendeten Build-Systemen hat das 1976 an den Bell Labs von Stuart Feldman entwickelte Unix `make` weitaus mehr als eine nur historische Bedeutung, in der Kernsyntax sind auch moderne Derivate wie GNU make und CMake immer noch identisch zu ihrem Vorbild.

Abhängigkeiten zwischen Header-Files

Nicht selten kommt es auch zu Abhängigkeiten von Header-Files untereinander, z.B. wenn

- in einem Header-File ein Datentyp oder eine Klasse **verwendet** wird,
- die in einem anderen Header-File **definiert** ist.

In solchen Fällen ist es meist üblich, im abhängigen Header-File den als Voraussetzung erforderlichen zweiten Header-File direkt zu inkludieren.

```
// header file: Base.h
class Base {
    …
};
```

```
// header file: Derived.h
#include "Base.h"
class Derived : public Base {
    …
};
```

Include-Guards

Da ein und derselbe Header-File oft auf verschiedenen Wegen inkludiert wird, muss die **mehrfache Verarbeitung*** ausgeschlossen werden. Dies geschieht mit sogenannten Include-Guards:

```
// header file: Base.h
#ifndef BASE_H
#define BASE_H
class Base {
    …
};
#endif
```

```
// header file: Derived.h
#ifndef DERIVED_H
#define DERIVED_H
#include "Base.h"
class Derived : public Base {
    …
};
#endif
```

*: Der Grund hierfür liegt vor allem in der One Definition Rule (ODR), welche die wiederholte Einführung von Bezeichnern stark einschränkt.

Namespaces

Wie das folgende Beispiel deutlich macht, ist der Klassenname für sich allein als Include-Guard u.U. problematisch:

```
#ifndef SOMECLASS_H
#define SOMECLASS_H
namespace Mine {
    class SomeClass {
        …
    };
}
#endif
```

```
#ifndef SOMECLASS_H
#define SOMECLASS_H
namespace Other {
    class SomeClass {
        …
    };
}
#endif
```

Wenn jetzt beide Files inkludiert werden, macht der erste den Inhalt des zweiten quasi "unsichtbar" ... was aber vermutlich erst nach einer überaus langwierigen Fehlersuche offenbar wird.

[!] Somit sollte der Include-Guard auch den Namen des `namespace` enthalten, also z.B. `MINE_SOMECLASS_H` und `OTHER_SOMECLASS_H`.*

*: Auch stellt sich die Frage, ob man im Include-Guard tatsächlich alle Buchstaben groß schreiben sollte ...

Zyklische Abhängigkeiten

Einiges Kopfzerbrechen dürfte die Fehlermeldung bereiten, welche trotz (oder wegen?) des Include Guard aus der folgenden Situation resultiert:*

```
// file: someclass.h
#ifndef SOMECLASS_H
#define SOMECLASS_H
#include "OtherClass.h"
…
class SomeClass {
    …
    OtherClass *link;
};
#endif
```

```
// file: otherclass.h
#ifndef OTHERCLASS_H
#define OTHERCLASS_H
#include "SomeClass.h"
…
class OtherClass {
    …
    SomeClass *link;
};
#endif
```

*: In kompilierbarer Form finden Sie die Dateien zu diesem Beispiel unter Examples/Cyclic/Broken und Examples/Cyclic/Fixed (fehlerbereinigt).

Definition/Reference-Modell

Zu jeder in einem Progamm verwendeten Variablen muss es **genau eine** Definition geben (mit möglicherweise vielen Bezugnahmen).

- Bei der Bezugnahme aus einer anderen Übersetzungseinheit muss diese eine `extern`-Deklaration vornehmen.
- Die tatsächliche Definition **darf** das Wort `extern` enthalten, auch wenn sie mit einer Initialisierung verbunden ist.
- Sie **muss** dies eventuell* sogar, wenn eine `const`-Qualifizierung verwendet wird und keine `extern`-Deklaration vorausgeht.

[!] Die Verwendung von `const` auf globaler Ebene **ohne** `extern` impliziert den Sichbarkeitsschutz gegenüber dem Linker.

*: "Eventuell" bedeutet hier, dass die Definition in einer Implementierungsdatei (`*.cpp`) vorgenommen wurde, aber **für andere Übersetzungseinheiten sichtbar** sein soll. Zum Hintergrund: Einstmals wurde von Bjarne Stroustrup damit das Ziel verfolgt, in Header-Files(!) eine möglichst unproblematische Umstellung von `#define`-s auf `const` zu unterstützen. Das Risiko von Name Clashes in der Link-Phase wurde mit der Regel ausgeschlossen, dass "`const` quasi `static` impliziert ... mit dem Nachteil, dass globale Konstanten evtl. mehrfach im Speicher gehalten werden. (Bei einfachen Datentypen können optimierende Compiler dies wiederum vermeiden, indem sie für solche Konstanten – solange deren Addresse nicht verwendet wird – überhaupt keinen Speicherplatz anlegen.)

Schreibschutz durch den Compiler (`const`)

Mittels `const`-Qualifizierung wird die Zuweisung eines (neuen) Werts an eine Variable verboten. Es ist nur noch die Initialisierung bei der Definition möglich bzw. ohne `extern` auch erforderlich.

```
const int x = 42;              // Initialisierung notwendig
extern const unsigned VERSION; // nur Bezugnahme
```

Folgendes führt nun zu einem Compile-Fehler:*

```
++x;
```

Oder auch:

```
if (VERSION = 3014u) {
    // special case for version 3.14
    …
}
```

*: Abhängig von der Art der Variablen und den Möglichkeiten der Hardware ist für `const`-qualifizierte Variablen eventuell auch ein physikalischer Schreibschutz möglich.

Zeiger und Referenzen

- Zeiger allgemein und `const`-qualifiziert
- Klassische Referenzen (Lvalue-Referenzen)
- Zeiger versus Referenzen
- Rvalue-Referenzen (neu in C++11)

Zeiger allgemein und `const`-qualifiziert

```
int *p1;        // pointer and pointed-to memory are modifiable
p1 = …;         // OK
*p1 = …;        // OK (assuming p1 points to valid memory now)
```

Bei Zeigern kann sich `const` **auf den Zeiger selbst** beziehen …

```
int *const p3 = …; // must be initialized (with address of an int)
p3 = …;            // ERROR (would modify pointer itself)
*p3 = …;           // OK (modifies pointed-to memory location)
++*p3;             // OK (increments pointed-to memory location)
++p3;              // ERROR (would increment pointer itself!)
```

… oder auf das, was **über den Zeiger erreichbar** ist:*

```
const int *p2; // same as: int const *p2;
p2 = …;        // pointer is still modifiable, but ...
*p2 = …;       // ... ERROR at compile-time!
```

---

*: Of course, both kinds of `const`-qualification may be combined if it makes sense for a given purpose, e.g. `const int *const p4 = …;` (or – switching positions of the qualification and the type to which it is applied: `int const *const p4 = …;`).

Klassische Referenzen (Lvalue-Referenzen)

Das klassische Beispiel ist eine Funktion, welche die Inhalte zweier Variablen vertauscht:

```
void swap(int *p, int *q) {
    const int t = *p;
    *p = *q;
    *q = t;
}
…
int a, b;
…
if (a > b) swap(&a, &b);
…
```

```
void swap(int &r, int &s) {
    const int t = r;
    r = s;
    s = t;
}
…
int a, b;
…
if (a > b) swap(a, b);
…
```

In der linken Version werden Zeigerargumente verwendet und explizit Adressen übergeben, in der rechten sind die Argumente Referenzen. Der Unterschied besteht aber nur in der sparsameren Notation im Quelltext (mit Referenzen) – es kann identischer Maschinencode erzeugt werden.*

*: Though there is no rule to enforce this. E.g. depending on compile time debug options different security checks could be generated for pointers and references.

Initialisierung von Referenzen

Da Referenzen konzeptionell stets vorhandenen Speicherplatz bezeichnen,* müssen sie zwingend initialisiert werden.

> Eine Referenz kann nach Definition und Intialisierung nicht mehr so verändert werden, dass sie auf eine andere Speicherstelle verweist.

Insofern entsprechen Referenzen den konstanten Zeigern, die selbst nicht veränderbar sind, wohl aber der über sie erreichbare Speicherplatz.

```
int v1, v2;  // some variables ...
int &r = v1; // rerefence is intialized
```

Das folgende führt **nicht** zu einem Fehler bei der Kompilierung, es führt aber auch nicht dazu, dass `r` nun die Variable `v2` referenziert:

```
r = v2;  // copies current content of v2 to v1 (referenced by r)
```

*: Not considering some artistic ways of initialisation to deliberately subvert this property of a reference (like `T &r = *reinterpret_cast<T*>(0);`), an invalid reference might be created by accident when a dereferenced pointer is used to initialize a reference without prior checking: `T *p = 0; … T &r = *p;`

Konstante Referenzen

Die `const`-Qualifizierung bezieht sich bei der Referenz auf den darüber möglichen Zugriff:

- Im Allgemeinen kann über eine Referenz angesprochener Speicherplatz gelesen und verändert werden.
- Über eine `const`-qualifizierte Referenz sind nur Lesezugriffe möglich.

Im folgenden Beispiel kann die Variable `v` zwar direkt und bei Zugriff über `r1` verändert werden, nicht aber beim Zugriff über `r2`:

```
T v;
T &r1 = v;       // r1 now refers to content of v
const T &r2 = v; // r2 also refers to content of v
…
r1 = …; // OK (actually changes v)
r2 = …; // ERROR (at compile time)
```

Lesender Zugriff ist natürlich sowohl über `r1` wie auch über `r2` möglich.*

*: It should be obvious that in the light of references a value-tracking compiler must be careful not to optimise-out *read* memory access with no intervening *write*: the content of some location might still have been modified through a different access path.

Achieving the Const-Correctness

The C++ compiler catches constructs that may subvert const-correctness.

```
const T cv = …;
const T &cr = cv; // OK
T &r = cv;        // ERROR
```

```
const T cv = …;
const T *cp = &cv; // OK
T *p = &cv;        // ERROR
```

Aside from the notational there is **no substantial difference** between pointers and references in the two examples above.*

[!] If the above initialisation would compile, the non-modifiable variable `cv` might be modified via a non-`const` reference (`r`) or a pointer to non-`const` memory (`p`).

*: GCC usually emits the same machine code for references and pointers, as long as both are used correctly and semantically equivalent. (To try some examples easily you may want to go to the following site: http://gcc.godbolt.org)

Reference Arguments

As reference initialisation occurs when handing arguments to functions, all the peculiarities and special cases discussed so far will be mostly likely observed there.

A typical lapse is to forget to add `const` to read-only reference arguments:

```
void foo(double &arg) {
    …
    … // all access to arg is non-modifying
    …
}
```

Not callable with literal constant or `const`-qualified variable:

```
foo(0.0); // ERROR
double const PI = 3.14;
foo(PI);  // ERROR
```

No temporaries are silently created:*

```
int x = 42;
foo(x);    // ERROR
foo(2*PI); // ERROR
```

*: The reason is to avoid surprising effects that would occur especially if a temporary were created for type coercion, and modifications were then applied to the temporary only but not to the variable actually used as argument (though it was "obviously" handed over by reference).

Zeiger versus Referenzen

C++ Referenzen können auf zwei Arten betrachtet werden:

- Eine alternative Syntax für Zeiger, welche
  - die Dereferenzierung bei der Verwendung impliziert (also "*" automatisch vorangestellt);
  - bzw. den Adress-Operator bei der Initialisierung (also "&" automatisch vorangestellt).
- Oder aber einen Alias-Name für bereits (an anderer Stelle) existierenden, typisierten Speicherplatz.

Der für Zeiger und Referenzen erzeugte Maschinen-Code unterscheidet sich in der Regel nicht – unterschiedlich ist nur die Syntax bei Initialisierung und beim Zugriff auf das, was referenziert wird.*

*: Der Nachweis ist bei `g++` durch Erzeugung des Assembler-Codes möglich, wozu die Option "`-S`" (Großbuchstabe) anzugeben und das Resultat dann in einer Datei mit Suffix "`.s`" (Kleinbuchstabe) zu finden ist.

Rvalue-Referenzen

Mit C+11 neu eingeführt wurde das Konzept der Rvalue-Referenzen. Sie lassen sich nur mit Ausdrücken initialisieren, also *temporären Werten*, auf die dann kein anderer Zugriff als über die Referenz besteht.

Nachfolgend zusammengefasst die wichtigsten Regeln:*

```
T &r = …;           // … must be modifiable T in memory
const T &cr = …;    // … must be modifiable T in memory OR
                    //       non-modifiable T in memory OR
                    //       temporary T in memory (expression)
T &&rr = …;         // … must be temporary T in memory (expression)
```

Die Hauptanwendung liegt beim Überladen von Funktionen für unterschiedliche Herkunft von Argumenten, und dort wiederum insbesondere bei Kopier-Konstruktor und -Zuweisung, denen damit Move-Varianten zur Seite gestellt werden können.

*: In dem für die obige Szenarien typischen Fall von Funktionsargumenten kann bei einer Rvalue-Referenz – anders als bei einer klassischen `const`-Referenz – das übergebene Argument modifiziert werden. Die Freiheit, dies zu tun, reicht allerdings nur so weit, dass der Destruktor nach wie vor korrekt funktionieren muss!

Statischer Polymorphismus

- Defaultwerte für Argumente
- Überladen von Funktionen
- Überladen von Operatoren

Defaultwerte für Argumente wurden in diesen Abschnitt aufgenommen, da ihre Wirkung mit dem Überladen von Funktionen konkurriert.*

*: Aus Hardware-Sicht ist die Implementierung bei echten Funktionen (ohne `inline`) allerdings unterschiedlich.

Defaultwerte für Argumente

Argumente können "von rechts nach links" mit Default-Werten versehen werden, das heißt:

- Sobald ein Argument einen Default-Wert hat,
- müssen alle rechts davon stehenden ebenfalls einen haben.

> Der Defaultwert muss beim Funktionsaufruf bekannt sein, ist also Bestandteil des Funktions-Prototyps und steht ggf. zusammen mit diesem in einem Header-File.

Die Namen für die formalen Argumente sind auch in diesem Fall optional:*

```
// the following function can be called with 1..3 arguments:
double foo(int &count, int minsize = 0, char separator = 'z');
…
// same as:
double foo(int &, int = 0, char = 'z');
```

*: Using names for arguments in prototypes has pro's and con's: it is of course more self-documenting but there is at least a remote chance for surprising and **extremely hard to find** name clashes with preprocessor macros. Hint: view preprocessor output (`g++ -E …`) whenever you get desparate because of an completely unexplainable syntax error in your source code.

Überladen von Funktionen

Mehrere Funktionen gleichen Namens können parallel existieren sofern sie sich in Anzahl und/oder Typ ihrer Argumente unterscheiden:*

```
void foo(const char *);          /*1*/
double foo(const int &, char);   /*2*/
double foo(double, double);      /*3*/
```

Welche Funktion aufgerufen wird (oder der Aufruf mehrdeutig ist), entscheidet der Compiler gemäß den tatsächlichen Argumenten.

```
int x; double y;
…
foo("hello, world");   /*calls 1*/
foo(42, 'z');          /*calls 2*/
foo(y, y/2);           /*calls 3*/
foo(x, y);             /*calls ?*/
```

*: If one argument list is from left to right an exact subset of another one, the overall effect is similar to default values for arguments. But with overloading there are as many separate entry points as there are functions, while with default arguments there is just one entry point and missing argument values are automatically supplied.

Überladen bei Parametern mit ohne `const`

Auch die `const`-Qualifizierung eines Parameters macht einen Unterschied:

In Fortsetzung des Beispiels von der vorhergehenden Seite:

```
char data[100];
…
foo(data); /*1*/
```

Mit **Erweiterung** dieses Beispiels:

```
void foo(char *); /*4*/
extern const char greet[];
…
foo(data);  /*4*/
foo(greet); /*1*/
```

Existiert nur eine der beiden Funktionen (die sich voneinander nur in der `const`-Qualifizierung eines Parameters unterscheiden), so gilt:

- Existiert **allein** die Funktion für den konstanten Fall, wird sie **auch** für modifizierbare Argumente verwendet.*
- **Ohne** die Funktion für den konstanten Fall führt der Aufruf mit einem nicht-modifizierbaren Argument zum **Compile-Fehler** (im Beispiel etwa `foo("hi")`, wenn nur Version `/*4'/` aber nicht `/*1*/` existiert).

*: Es ist kein Problem einer Funktion, die verspricht, den Inhalt einer über ein Zeiger- oder Referenz-Argument erreichbaren Variablen nicht zu verändern, Zugriff auf (prinzipiell) veränderbaren Speicherplatz zu geben – **wohl aber umgekehrt!**

Überladen von Operatoren

Operatoren* können überladen werden mit Funktionen, deren Name mit dem Wort `operator` beginnt.

- Die meisten Operatoren können wahlweise mit freistehenden Funktionen oder mit Member-Funktionen überladen werden.
- An der Überladung muss aber mindestens eine Klasse beteiligt sein.*
- Einige Operatoren sind hinsichtlich der Überladung auf Member-Funktionen eingeschränkt.
- Zur konsistenten Überladung ganzer Operatorgruppen kann Boost.Operators hilfreich sein.

Weiterführende Links:

- http://en.cppreference.com/w/cpp/language/operators
- http://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

*: Therefore the meaning of operators for builtin-types cannot be changed. If you want to come close to the behaviour of a given builtin types but change or remove some predefine operations, you will typically have to **add** a new class and implement all the operations it should support.

Operator-Überladung mit freistehender Funktion

Diese sieht prinzipiell so aus:

```
MyClass operator+(const MyClass &lhs, const MyClass &rhs) {
    … // do whatever must be done
    return …;
}
```

Der Rückgabetyp ist dabei beliebig, die `return`-Anweisung muss natürlich vom Typ her passend sein.

Genauer gesagt, der Ausdruck hinter `return` muss

- entweder exakt den Rückgabetyp haben (`MyClass` im Beispiel)
- oder in diesen umwandelbar sein.

Operator-Überladung mit Member-Funktion

Diese sieht prinzipiell so aus:

```
MyClass &MyClass::operator+=(const MyClass &rhs) {
    … // do whatever must be done
    return *this;
}
```

Auch hier ist der Rückgabetyp grundsätzlich frei wählbar.

> Gemäß den Konventionen bei eingebauten Typen wird in der Regel das durch die Operation gerade veränderte Objekt selbst zurückgegeben.

Die Rückgabe erfolgt typischerweise als Referenz (sonst wäre es auch nicht das Objekt selbst sondern nur eine identische Kopie).

Da dies technisch gesehen nur Weitergabe einer Adresse bedeutet, ist es

- performant (z.B. Rückgabe in Register) und
- nahezu frei von Overhead bei Nichtbenutzung*

*: Also, for an optimizing compiler there is a fair chance to remove any remaining overhead ...

Überladung von Kopier-Konstruktor und -Zuweisung

Für einige Arten von Objekten müssen diese Operationen überladen werden, da der Default – elementweises Initialisieren bzw. Zuweisung – ungeeignet ist.*

```
class MyClass {
    T *some_ptr;
    …
public:
    …
    // avoid compiler defaults:
    MyClass(const MyClass& rhs);
    MyClass& operator=(const MyClass& rhs);
}
```

Der klassische Indikator sind als Member-Daten enthaltene Zeiger auf Speicherplatz, welcher individuell für jedes Objekt vorhanden sein muss.*

*: In C++ books this is often referred to as Rule of Three – the third member function for which the default is not appropriate is the destructor, of course.

Überladung von Move-Konstruktor und -Zuweisung

In C++11 können die Rvalue-Referenzen dazu verwendet werden, abhängig davon, ob der dabei als Operand auftretende Ausdruck

- direkt ein Objekt repräsentiert, das danach unverändert im Speicher weiter existiert, oder
- temporären Speicherplatz, der ohnehin im Anschluss verworfen wird (z.B. für einen berechneten Ausdruck oder ein Funktionsergebnis),

Initialisierung wie auch Zuweisung unterschiedlich zu implementieren.*

```
class MyClass {
public:
    …
    // copy versions (rhs lives on in memory)
    MyClass(const MyClass& rhs);
    MyClass& operator=(const MyClass& rhs);
    // move versions (rhs destroyed soon after)
    MyClass(MyClass&& rhs);
    MyClass& operator=(MyClass&& rhs);
}
```

*: Turning the classic C++ *Rule of Three* into the Rule of Five in C++11.

Implementierung von Move-Konstruktor/Zuweisung

Nachdem Move-Versionen deklariert sind, müssen diese natürlich auch implementiert werden.

Wenn der Kopierkonstruktor wie folgt aussieht …

```
MyClass::MyClass(const MyClass &rhs)
    : …, some_ptr(new T(*rhs.some_ptr)), …    // cloning resource
{ … }
```

… könnte dieser Move-Konstruktor angemessen sein:

```
MyClass::MyClass(MyClass &&rhs)
    : …, some_ptr(rhs.some_ptr), …    // taking over resource
{ …; rhs.some_ptr = nullptr; … }      // INVALIDATING it for rhs!
```

Die Zuweisungen sind ähnlich, müssen aber zuerst `some_ptr` freigeben.*

*: The general difference between constructor and assignment is that the former gets just a piece of memory while the later finds a valid object that needs to be properly de-constructed first.

Unterscheidung Copy- und Move-Versionen

Existieren beide Fassungen (Copy und Move), ergibt sich folgendes Verhalten:*

```
MyClass foo() { return …; }  // … must be an expression of
                             // type MyClass (or something
                             // convertible to MyClass)
// constructor use:
MyClass a;        // (expects c'tor with no argumens)
MyClass b(a);     // copy c'tor (does not modify a)
MyClass c(foo()); // move c'tor (may modify temporary)

// assignment use:
a = c;            // copy assignment (does not modify c)
b = foo();        // move assignment (may modify temporary)
c = a + b;        // move assignment (may modify temporary
                  //                  returned from operator+)
```

*: Of course, adding operands of type `MyClass` in the last line of the example also assumes `operator+` exists and returns by value, as is the usual behaviour. In the (unusual) case that operator were defined but returns something that C++ considers to be "more persistant" (like a reference), copy assignment would be used instead, though a move could be enforced then: `c = std::move( … );`

Typ-Kompatibilität und -Konvertierungen

- Typ-Kompatibilität bei grundlegenden Typen
- Typ-Kompatibilität und Vererbung
- Explizite Typ-Konvertierung mittels *Cast*
- Klassenspezifische Typ-Konvertierung
- Typ-Sicherheit generell

Typ-Kompatibilität bei grundlegenden Typen

Die wichtigsten Regeln für Typ-Kompatibilität und ggf. automatisch angewendete Typ-Konvertierungen sind hier:

- Alle *arithmetischen Typen* sind miteinander kompatibel bzw. werden bei Bedarf entsprechend umgewandelt.*
- Ansonsten finden bei Bedarf folgende Umwandlungen statt:
  - Aufzählungstypen (`enum`) → arithmetische Typen;
  - Zeiger → Wahrheitswerte (alles außer `nullptr` ist `true`);
  - typisierter Zeiger → allgemeine Zeiger (`void *`).

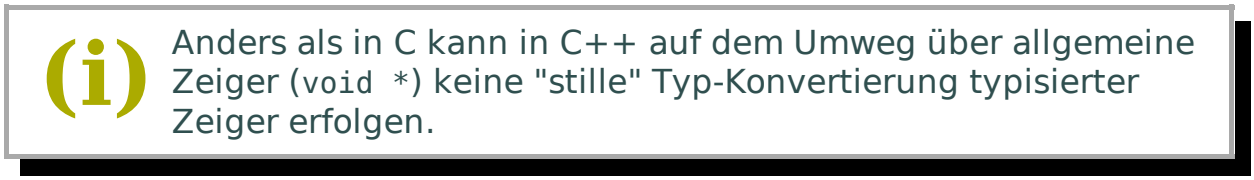
Inwieweit dies auch für die anderen, oben aufgezählten Fälle gilt, hängt von Hardware-Details ab, insbesondere der Repräsentation von Aufzählungstypen und Zeigern.

*: Dies schließt auch `char` (mitunter verwendet für kleine Ganzzahlen) und `bool` (Wahrheitswerte) ein.

Typ-Kompatibilität bei grundlegenden Typen (2)

- Außerhalb der Umwandlungen zwischen arithmetischen Typen handelt es sich um **einseitig gerichtete** Umwandlungen.
- D.h. es gibt **keine automatische Umwandlung** in den folgenden Fällen:
  - von arithmetischen Typen in Aufzählungstypen (`enum`);
  - von Wahrheitswerten in Zeiger;
  - **von allgemeinen Zeigern in typisierte Zeiger**.

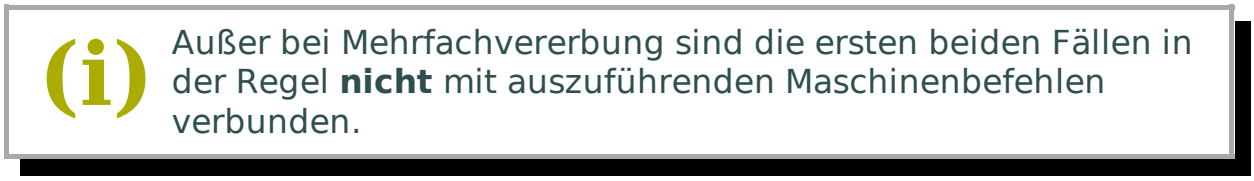
Mit dem letzten Punkt wurde die Typ-Sicherheit von C++ gegenüber C an einer kritischen Stelle verbessert.*



*: Der Grund für das andere Verhalten in C liegt bei der `malloc`-Funktionsfamilie in der C-Bibliothek, die gemäß dem C89-Standard den Ergebnistyp `void *` hat. In C wollte man so die die Notwendigkeit von Cast-Operationen für `malloc` etc. vermeiden. Da in C++ `new` ein Operator ist, entfällt diese Problematik.

Typ-Kompatibilität und Vererbung

- Ein **Zeiger** auf eine öffentlich abgeleitete Klasse ist kompatibel mit einem **Zeiger** auf eine ihrer direkten oder indirekten* Basisklassen.
- Ein **Referenz** auf eine öffentlich abgeleitete Klasse ist kompatibel mit einer **Referenz** auf eine ihrer direkten oder indirekten* Basisklassen.
- Basierend auf **Objekten** einer öffentlich abgeleiteten Klasse werden ggf. automatisch – per Slicing – **Objekte** direkter oder indirekter* Basisklassen initialisiert.



*: Für indirekte nicht-virtuelle Basisklassen im Fall von Mehrfachvererbung mit rautenförmigen Klassenbeziehungen (in der UML als "«disjoint»" annotiert) gilt dies nicht, da in diesem Fall die automatische Konvertierung mehrdeutig wäre.

Öffentliche Basisklassen und LSP

Die auf der vorhergehenden Seite zusammengestellten Regeln sind Ausdruck dessen, was Barbara Liskov seinerzeit als

- "*Principle of Substitutibility*"

für die Objekt-Orientierte Programmierung als forderte und was seitdem oft als LSP abgekürzt wird.

> Das LSP gilt in C++ nur im Fall öffentlicher Basisklassen.
>
> **Nur nur hier liegt Vererbung im Sinne der OOP vor.**

Vererbung* wird auch *Generalisierung-Spezialisierung* genannt und in der UML-Darstellung durch eine Verbindungslinie mit nicht ausgefülltem Dreieck am auf die Basisklasse weisenden Ende spezifiziert.

---

*: To be clear once more: what is discussed here is a class relation that implies substitutibility according to the LSP. As inheritance is at the heart of object-oriented modelling and programming, it will be covered in more depth later (see Wednesday Part 1 Inheritance).

Private Basisklassen (kein LSP)

Öffentliche und private Basisklassen in C++

- verwenden zwar ein und dasselbe Speicher-Layout,
- **bei privaten Basisklassen gilt das LSP jedoch nicht**.

> Im Sine der OOP handelt es sich bei letzteren somit nicht um Vererbung sondern um Komposition.

Komposition ist ein Sonderfall der Aggregation* und wird in der UML-Darstellung durch eine Verbindungslinie mit ausgefüllter Raute am auf die Klasse des Aggregats weisenden Ende spezifiziert.

*: The special case is that the lifetime of the *part* is coupled to that of the aggregate. As composition is very important in object-oriented modelling and there are various ways to implement it in C++, it will be covered in more depth later (see Wednesday Part 1 Aggregation).

Typ-Konvertierung mittels *Cast*

Mittels sogenannter Cast-Operationen lassen sich weitere Typ-Konvertierungen erzwingen.

> **[!]** Die Cast-Syntax von C, bei welcher man den Zieltyp in runde Klammern einschließt und den umzuwandelnden Wert direkt dahinter schreibt, sollte in C++ vermieden werden.

Die neue Syntax beginnt mit einem der Schlüsselworte

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

Es folgen der Zieltyp in spitzen Klammern und der umzuwandelnde Wert in runden Klammern.*

---

*: As an example consider the use of C-style memory allocation for some `struct s` in C++.

```
struct s *p = static_cast<struct s*>(std::malloc(sizeof (struct s))); // required conversion in C++
            … (struct s*) std::malloc(sizeof (struct s)); // C-style cast, possible but deprecated in C++
```

Typ-Konvertierung mit `static_cast`

Hiermit lassen sich zum einen Typ-Konvertierungen explizit hervorheben, welche der Compiler auch automatisch vorgenommen hätte.*

Darüberhinaus funktioniert der `static_cast` in **beide** Richtungen für diejenigen Typ-Konvertierungen, welche **automatisch** nur in einer Richtung eingesetzt werden:

- Arithmetische Wert → Aufzählungstypen (`enum`)
  - automatisch nur Aufzählungstypen in arithmetische Werte
- Generische Zeiger (`void *`) → typisierte Zeigern
  - automatisch nur typisierte in generische Zeiger
- Basisklassen in abgeleitete Klassen (Downcast)
  - automatisch nur abgeleitete Klassen → Basisklassen (Upcast)

*: The most typical reason for this is that many compilers will issue a warning when an arithmetic conversions might not be value preserving, e.g. when an 64-bit integral value is assigned to a 32 bit integral variable. If this is done with a cast, most compilers will suppress the warning.

Typ-Konvertierung mit `dynamic_cast`

Hiermit lassen sich ausschließlich Typ-Konvertierungen innerhalb von Klassenhierarchien vornehmen.

> Im Fall von Downcasts findet zur Laufzeit eine Überprüfung mit Fehleranzeige statt, wenn der Cast nicht durchführbar ist.*

Die Fehleranzeige besteht

- bei **Casts auf Zeigerbasis** in der Rückgabe eines Nullzeigers;
- bei **Casts auf Referenzbasis** in einer `std::bad_cast`-Exception.

Weiteres wird später im Rahmen der Laufzeit-Typprüfung (RTTI) behandelt.

---

*: Der Grund für die eventuelle Undurchführbarkeit muss zusammen mit der Typ-Kompatibilität zwischen Basisklassen und abgeleiteten Klassen gesehen werden: Gemäß LSP kann ein Zeiger oder eine Referenz, der bzw. die als Zeiger oder Referenz für eine Basisklasse definiert ist, auch ein Objekt einer davon abgeleiteten Klasse referenzieren. Ob dies der Fall ist, lässt sich mit `dynamic_cast` überprüfen und mittels des auf diese Weise ggf. erhaltenen Zeigers (bzw. der so erhaltenen Referenz) besteht schließlich Zugriff auf die von der abgeleiteten Klasse hinzugefügten Member-Daten und -Funktionen.

Typ-Konvertierung mit `const_cast`

Die hiermit möglichen Typ-Konvertierungen beschränken sich auf das

- Hinzufügen oder
- Wegnehmen

von `const` und `volatile`.

Alle anderen Unterschiede zwischen dem Zieltyp und dem Typ des umzuwandelnden Ausdrucks führen zu einem Compile-Fehler.

> **[!]** Ein `const_cast` hat gemäß dem C++-Standard undefiniertes Verhalten, wenn er dazu führt, dass auf eine mit Schreibschutz definierte Speicher-Adresse schreibend zugegriffen wird.

Das typische Fehlerbild reicht von der inkonsistenten Wertverwendung (teilweise alter Wert, teilweise neuer Wert) bis zum Programmabsturz ...*

*: Abhängig von der Testtiefe, dem verwendeten Compiler, Optimierungs-Optionen usw. mag es allerdings auch so erscheinen, als würde alles wie gewünscht funktionieren.

Typ-Konvertierung mit `reinterpret_cast`

Dieses Konstrukt wird vor allem dazu eingesetzt, Zeiger auf (bekannte) Hardware-Adressen zu setzen, wie das u.a. im Bereich der Embedded Progammierung und bei Gerätetreibern notwendig sein kann.*

Darüber hinaus kann man mit einem `reinterpret_cast`

- **wie auch per `static_cast`** generische Zeiger (`void *`) in typisierte Zeiger umwandeln, und
- **anders als per `static_cast`** typisierte Zeiger **direkt** in anders typisierte Zeiger umwandeln.

> Es ist dagegen auch mit `reinterpret_cast` nicht möglich, die `const`- und `volatile`-Qualifizierung zu ändern – dies erlaubt nur der `const_cast`.

Um beide Konvertierungen zu kombinieren, müssen die jeweiligen Cast-Konstrukte ggf. nacheinander (oder geschachtelt) verwendet werden.

---

*: Dass es per `reinterpret_cast` möglich ist, quasi jedes Bitmuster im Speicher gemäß jedem beliebigen Typ zu interpretieren, führt mitunter zu der Kritik, C++ sei keine "sichere" Programmiersprache. Diese Kritik müsste dann aber für alle Sprachen gelten, die ein zur C/C++ `union` vergleichbares Konstrukt besitzen ... (zumindest solange kein automatisches "type-tagging" wie bei Boost.Variant erfolgt).

Typ-Konvertierung mit `reinterpret_cast` (Beispiel 1)

Das folgende Beispiel nimmt an, dass an der Speicheradresse `0xEAD0` die als `struct uart` beschriebenen Kontrollregister abgebildet sind:

```
struct uart *const sio = reinterpret_cast<struct uart*>(0xEAD0);
```

Der Zugriff kann nun in der Syntax `sio->…` erfolgen.

Gibt es mehrere solche Register-Strukturen im Speicher (als Memory-Mapped-I/O) abgebildet, kann natürlich auch ein Array initialisiert

```
struct uart *const sio[] = {
    reinterpret_cast<struct uart*>(0xEAD0),
    reinterpret_cast<struct uart*>(0xEAD8),
    …
};
```

und über dieses mit `sio[0]->…`, `sio[1]->…` usw. zugegriffen werden.

---

*: Wird statt dem Pfeil die Punkt-Notation bevorzugt, geht das auch, und zwar mit:

```
struct uart &sio = *reinterpret_cast<struct uart*>(0xEAD0);
```

Typ-Konvertierung mit `reinterpret_cast` (Beispiel 2)

Das folgende Beispiel nimmt an, dass an der Adresse `0xCAFE` im Code-Segment

- ein Unterprogramm (gemäß C++ Calling Conventions) steht,
- welches ein Argument vom Typ `bool` erwartet und
- keinen Rückgabewert liefert.

```
void (*xcall)(int) = reinterpret_cast<void (*)(int)>(0xCAFE);
```

Der tatsächliche Aufruf kann nun so erfolgen:

```
xcall(true);
…
xcall(false);
```

erfolgen.*

*: Of course, if there is a value returned supplied (according to the C++ calling conventions) and specified in the declaration, it could be accessed in the usual way.

Klassenspezifische Typ-Konvertierungen

Eine Klasse kann auch festlegen, wie ihre Objekte bei Bedarf automatisch **aus** eingebauten Typen und anderen Klassen erzeugt bzw. **in** solche konvertiert werden können. Dazu folgende Analogie:

- Jede Klasse und jeder eingebaute Typ hat eine spezifische Art von *Ein- und Ausgangs-Steckverbindern*, die nur "zu sich selbst" passt.
  - In Initialisierungen und Zuweisungen sind daher zunächst nur Objekte von genau dieser Klasse bzw. diesem Typ verwendbar.._[]
- **Konstruktoren mit exakt einem Argument** – sofern nicht als `explicit` markiert – sind weitere *Eingangs*-Steckverbinder.
  - Sie passen zum *Ausgangs*-Steckverbinder des Argument-Typs.
- **Typ-Cast-Operatoren** sind weitere *Ausgangs*-Steckverbinder.
  - Sie passen zum *Eingangs*-Steckverbinder des Ziel-Typs.

---

*: If you like that picture you may include base class conversions by assuming plugs and sockets with the same basic shape for class hierarchies, using code pins to make the output connector of a derived class fit into the input connector of its base class(es), but not vice versa. (For standard conversions of basic types assume a set of adapter plugs that are applied as necessary.)

Typ-Konvertierungen durch Konstruktoren

Konstruktoren sind dann automatische Typ-Konvertierungen, wenn sie

- genau ein Argument besitzen und
- **nicht** mit dem Schlüsselwort `explicit` markiert sind.

```
class MyClass {
    …
public:
    MyClass(int);  // each single argument c'tor is an
                   // automatic conversion ... except
    explicit MyClass(double);  // it is marked explicit
    …
};
```

Anwendung von Konstruktoren zur Typ-Konvertierung

Typ-Konvertierungen durch Konstruktoren kommen wie folgt zur Anwendung:

```
void foo(MyClass);
…
foo(33);             // OK (automatic conversion by c'tor)
foo(3.3)             // ERROR, c'tor is explict
foo(MyClass(3.3));   // OK (c'tor is used explicitly)
…
MyClass x(-1);       // this also work with explicit c'tor but ...
x = 33;              // ... here an automatic conversion is necessary ...
x = 3.3;             // ... so this will fail if there is none
```

(i) Die Notwendigkeit einer Typ-Konvertierung bei der Zuweisung hängt auch davon ab, welche zusätzlichen Zuweisung-Operatoren eine Klasse ggf. definiert*

*: An assignment operator taking **exactly** type of the expression on its right hand side as argument will be always be preferred (and of course applied without any need for a conversion). In case the assignment operator takes a `const MyClass&` argument – as the automatically defined assignment does–, a temporary will be if there is a non-explicit c'tor or a type-cast operator applicable as automatic conversion.

Temporäre Objekte im Rahmen der Typ-Konvertierung

Bei Referenzübergabe ist zusätzlich zu beachten, dass bei der Konvertierung ein temporäres Objekt notwendig wird:

```
void baz(MyClass &); // non-const reference argument
…
baz(42);                        // ERROR (no automatic temporary
                                //        for non-const reference)
baz(MyClass(42));               // ERROR (c'tor call does not bind
                                //        to non-const reference)
{ MyClass tmp(42); baz(tmp); } // OK
```

Automatisch wird dies nur für `const`-qualifizierte Referenzen erzeugt:

```
void bar(const MyClass &); // const reference argument
…
bar(42);                        // OK (automatic temporary)
```

Typ-Konvertierung durch Type-Cast Operatoren

Type-Cast Operatoren benutzen eine spezielle Syntax, bei der **nach** dem Schlüsselwort `operator` der Zieltyp folgt:*

```
class MyClass {
    …
public:
    // this is called type-cast operator:
    operator Other() const { …; return …; }
                                    // ^-- Other (or at least
                                    //     convertible to Other)

    // the usual explicit alternative:
    int to_int() const { …; return …; }
                                  // ^-- int (or at least
                                  //     convertible to int)
};
```

*: Dieser stellt zugleich den Ergebnistyp dar, den die `return`-Anweisung liefern muss.

Automatische Anwendung von Type-Cast Operatoren

Die Typ-Konvertierungen von der vorhergehenden Seite kommen wie folgt zur Anwendung:

```
void foo(Other);
void bar(int);
…
MyClass m;
foo(m);           // OK, implicit use of type-cast operator
bar(m);           // ERROR (of course), but ...
bar(m.to_int());  // ... usual style for explicit conversion
```

Sonderfall: `explicit operator bool()`

Eine als `explicit` markierte Typ-Konvertierung in einen Wahrheitswert stellt einen Sonderfall dar:

- Sie kommt **nicht** zur Anwendung bei Argumentübergabe, Initialisierung und Zuweisung,
- **jedoch bei bool'schen Operationen und Bedingungstests.**

Die Beispiele auf der nächsten Seite setzen folgendes voraus:

```
class MyClass {
    …
public:
    explicit operator bool() {
       return …; // some bool
    }
    …
};
…
MyClass obj;
extern void foo(bool);
```

Beispiele: `explicit operator bool()`

Die folgenen Code-Fragmente setzen das begonnene Beispiel fort:

- `#1` zeigt eine etwas ungewöhnliche aber erlaubte Form des Aufrufs der Typ-Konvertierung in `bool`.
- `#2a` löst einen Fehler aus, der in einer syntaktische Mehrdeutigkeit begründet ist, die das Paar zusätzlicher Klammern in `#2b` beseitigt.
- `#3` ist eine zulässige aber überflüssige explizite Typ-Konvertierung.

```
// this does NOT compile ...
foo(obj);

bool bv(obj);
bool bv(bool(obj));      //#2a
bv = obj;

if (obj == true) …
if (obj == false) …
if (obj == bv) …
// ... compare with code on
// the right for corrections
```

```
// this solves the problems:
foo(bool(obj));
foo(obj.operator bool()); //#1

bool bv((bool(obj)));     //#2b
bv = bool(obj);
if (bool(obj)) …          //#3
if (obj) …
if (!obj) …
if (bool(obj) == bv) …
// boolean operators work too:
if (obj && !bv) …
```

Typ-Sicherheit in C++

Die praktische Konsequenz aus den Risiken, welche die Konstrukte zur expliziten Typ-Konvertierung – also die vier neuen Cast-Formen von C++ sowie die von C übernommene Cast-Syntax – mit sich bringen, ist diese:

> **[!]** Alle Formen expliziter Typ-Konvertierung (mit Cast) sollten auf das unvermeidliche Minimum beschränkt bleiben.

Ferner werden klassenspezifische Typ-Konvertierungen manchmal in "unerwarteter Weise" angewendet:*

> **(i)** Eine klassenspezifische Typ-Konvertierung sollte nur dort definiert werden, wo der stillschweigende Wechsel zwischen den beteiligten Typen als "natürlich" empfunden wird.

*: Or to put it slightly different: Experience showed there are scenarios of practical importance where a compile error would have been preferred over the way the compiler made the code "correct" by applying a (non-explicit) constructor or type-cast operator.

Praktikum