C++ FOR (Monday Morning)

Together with the following chapter this constitutes roughly what is planned to cover on Monday. Depending on the depth of coverage all these topics may even require more time than one day. This is especially the case if both is desired, refreshing traditional C++ knowledge **and** learning about what is new in C++11 and C++14.

Refreshing some C++ Basics

- Separate Compilation
- Definition/Reference Model
- Compile Time Write Protection

Some topics covered here were not part of C++98 but introduced with C++11.

> (i) If you have good knowledge of C++98 it may still make sense to give the following pages a cursory look and examine features introduced with C++11.

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`).

> **(i)** Bereits bei einer kleinen Zahl von Übersetzungseinheiten sind die Abhängigkeiten zwischen diesen inklusive ihrer Header-Files oft nur noch schwer zu überblicken, so dass sich die Verwendung eines Build-Systems empfiehlt.*

*: 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 Sichtbarkeitsschutz 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 Adresse 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.

Builtin and Standard Types

- Basic Integral Types
- Integral Type Standard Aliases
- Floating Point Types
- `void`, `bool`, and `enum`
- Querying Type Properties

Basic Integral Types

Built-in to C++ – i.e. available as reserved word without including any header file – are the following integral standard types:

- `char` / `signed char` / `unsigned char` / `wchar_t` / `char16_t` / `char32_t`
- `short` (same as `signed short`) / `unsigned short`
- `int` (same as `signed int`) / `unsigned int`
- `long` (same as `signed long`) / `unsigned long`
- `long long` (same as `signed long long`) / `unsigned long long`

Each of the above type names constitutes a type on its own.

Overloading may be based on these types – i.e. they are distinguishable – even if one has identical properties (signedness, limits, ...) as another one.

For more information on basic integral types (incl. character types) see subsections *Character Types* and *Properties* in: http://en.cppreference.com/w/cpp/language/types

*: E.g. by definition `char` has the same properties as either `signed char` or `unsigned char` (implementation defined). Also type `int` often has the same properties as either `short` or `long`.

Integral Type Standard Aliases

Header file `<cstdint>` centralises type definitions (aliases) of frequently required integral types with a specific number of bits.

Standard Aliases for Exact Width

These use name patterns like `int8_t`, `uint8_t`, `int16_t`, `uint16_t` ... `uint64_t` and these will only be defined if there is hardware support for **exactly** that number of bits.

Standard Aliases for Minimal Size

These use name patterns like `int_least8_t` ... `uint_least64_t` and guarantee the **smallest** integral type with at least that number of bits.

Standard Aliases for Fast Access

These use name patterns like `int_fast8_t` ... `uint_fast64_t` and guarantee the **fastest** integral type with at least that number of bits.

Floating Point Types

The three different type names `float`, `double` and `long double` are available for floating point.

- Often – especially if there is specific hardware support on the target platform – these will be mapped to the ISO754 types with 32, 64, and 80 bits.
- But the standard imposes no such requirement.
- In fact, a conforming implementation might even map all three types to the same hardware representation with only minimal requirements to range and accuracy.

For more information on floating point types see subsections *Floating Point Types* and table *Range of Values* in: http://en.cppreference.com/w/cpp/language/types

`void`, `bool`, and `enum`

These types serve different purposes which will be outlined in more detail on the following pages.

- `void`
  Denotes a type **with no valid values** and is used as result type of functions that return no results (i.e. classic subroutines).*
- `bool`
  Denotes a type with the only two values `true` and `false`.
- `enum`
  Denotes a type with a given set of values and was;
- `enum class`
  Denotes a variant of `enum`-s with some more features, introduced with C++11.

---

*: Note that `void *` has a different purpose, which is to define pointers with an unspecified type, i.e. "just addresses". It is often used for a certain form of "generic libraries", where mostly memory addresses are handed around, while information about the types really stored at these locations is buried in the program logic. An improved solution is Boost.Any, which encapsulates the type information with the pointer, making these sort of generic programming better readable (and a bit more secure).

Not Existing Values: `void`

The typical use of `void` is as return type of functions returning nothing (via their call name). Technically it is an [incomplete type], which means

- it cannot be part of (most sorts of) expressions,
- nor as type of function parameters,
- nor to define variables or class member data,
- ...

What is acceptable is to call a function returning `void` as part of the `return` statement of some other function:

```
void f() { … };
void g() { return f(); }
void h() { return g(); }
```

---

*: One realistic use case for this is in templated code, where `h` forwards to `g`, which may in turn forward to `f`, and it is expected all these functions have the same return type. Then this alleviates the need to have specialisation if the functions are generalised on their return type and this might be `void`:

```
// a set of type generic functions forwarding from one to the other
template<typename T> T f() { … }
template<typename T> T g() { … ; return f(); }
template<typename T> T f() { … ; return g(); }
```

Truth Values: `bool`

Because of the traditional type conversions applied to variables and expressions of type `bool`, it mixes seamlessly with all arithmetic types.

When using an expression in a context expecting a `bool`:

- any pointer other that `nullptr` converts to `true`;
- any arithmetic value other than zero converts to `true`;
- only zero and `nullptr` convert to `false`.

When using an expression of type `bool` in an arithmetic expression:

- the value `false` converts to `0` and
- the value `true` converts to `1` (in the specific type).

Note that type `bool` is usually represented **internally** as type larger than only one bit, as otherwise memory access would be inefficient.

Packed 0/1 Values

For dense "packing" of many 0/1-values, a classic array of `bool` is not space efficient:

```
bool bits[100000000];  // 100 million 0/1 values
```

In this case an `std::bitset` offers a trade-off that may be worth to consider:*

```
std::bitset<100000000> bits; // less space, more access time
```

*: For packed bit collections of a size which can only be determined at compile time, `std::vector<bool>` – a specialisation of `std::vector` – may be used. As this has its own set of potential problems, a detailed discussion will be postponed until later.

Classic Enumerations

Classic enums (as C++ inherited from C) have the following shortcomings:

- The names of the values declared by the `enum` type (aka. enum labels) are **imported into the enclosing namespace**,_[]
  - thus introducing the risking inadvertent name clashes, which is
  - usually reduced by scoping `enum`-s in the body of some `class`
- The **underlying type cannot be controlled** ...
  - ... at least not easily and ...
  - ... not in an implementation-dependant way.
- There **cannot be a forward declaration**,
  - e.g. in case the enum label values are not used but only its type as argument to a function
  - eventually resulting in (needless) recompilations when enum labels are added or changed in value.

*: Sometimes this specific property may not be a problem but an advantage.

C++11 Scoped Enumerations

With `enum class` C++11 resolved the problems of classic `enum`-s:*

```
enum class Color : unsigned short { Red, Green, Blue };
```

- The names of the labels are scoped:
  `Color::Red`, `Color::Green`, `Color::Blue`.
- The type can be easily controlled:
  `unsigned short` in this case (if not specified default is `int`).
- Forward declarations are possible:
  `enum class Color : unsigned short;`

*: Compared to enum classes in Java, C++ `enum class` is still poor on features!

Querying Type Properties

Header file `<limits>` defines specialisations of class `std::numeric_limits` for all arithmetic types (integral and floating) and also `bool`, providing tests for many properties of these type.

One typical use is to find the range of values representable in some type:

```
std::numeric_limits<short>::min()          // lowest short value
std::numeric_limits<short>::max()          // highest short value

std::numeric_limits<int>::min()            // same for int
std::numeric_limits<int>::max()            // etc.

std::numeric_limits<unsigned long>::min() // same for
…                                          // unsigned long etc.
```

*: Besides `std::numeric_limits` this header also defines the classes `std::float_round_style` and `std::float_denorm_style` specifying rounding and de-normalisation modes of floating point types.

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 Initialisierung nicht mehr so verändert werden, dass sie auf eine andere Speicherstelle verweist.

Insofern entsprechen Referenzen 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 initialise 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.

> Über eine `const`-qualifizierte Referenz sind nur Lesezugriffe möglich, selbst wenn die referenzierte Variable direkt auch veränderbar ist.

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!

Variable and Object Initialisation

- Classic Initialisation Syntax
- C++11 Brace Initialisation
- C++11 Direct Member Initialisation
- C++11 Compile Time Initialisation

Classic Initialisation Syntax

Until C++11 there were three different styles for initialisations:

- Assignment-like Syntax
- Constructor Call Syntax
- C (inherited) Syntax for Aggregates

There are also clearly defined rules what happens in case of:

- Uninitialised Variables
- Uninitialised Objects

Assignment-like Initialisation

The assignment-like uses an equals sign to separate the name of the variable (or object) initialised from the initialising expression.

```
int x = 42;
double pi = 3.14152;
std::string greet = "hello, world";
```

The difference to assignment is the type name on the left.*

*: Since C++11 the type name can also be replaced with `auto`, which means that the variable will have the type of the initialising expression, with reference (`&`), `const` or `volatile` qualifiers stripped away.

Constructor Call Syntax

Here the initialisation looks like calling a constructor, by following the object with a comma-separated list of argument values, enclosed in parentheses:

```
std::string greet("hello");
std::string line(80, '-');
```

An exception from this rule is initialising with the default constructor. In this case there are no parentheses following.

```
std::string empty;  // initialised with default (no content)
```

If (empty) parenthesis were used, this would be seen (like in C) as an `extern` declaration of a function returning an `std::string`:*

```
std::string foo(); // NOT a default-initialised std::string !!
```

*: If later that name were used with an `std::string` member function like `foo.empty()` the compiler will typically complain that it is expecting a function call operation.

C (inherited) Syntax for Aggregates

Here after the equals signed follow several comma separated values, enclosed in braces. This can be used as well for arrays*

```
int primes[] = {2, 3, 5, 7, 11};
```

as for structures

```
struct Point { float x, y; };
…
Point origin = {0.0, 0.0};
```

or combinations:

```
Point polygon[] = {
    { 7.23, 12.1 },
    { 1, 3.5  },
    …
};
```

*: If the size is omitted it will be determined from the number of initialising values.

Uninitialised Variables (of Basic Types)

Variables of basic types must not be initialised. Their initial value depends on the location of their definition:

Variables at a fixed address assigned at compile time, i.e.

- global variables,
- block-local `static` variables, and
- `static` class members

assume an initial value of

- `false` for booleans,
- `nullptr` for pointers, and
- zero (according to its type).

[!] Variables on the stack and on the heap have undefined initial values, maybe even "impossible" values for the given type on the given hardware.*

*: The content is usually just the "bits and bytes" previously stored at that memory location. On some hardware not all bit patterns are valid content for floating point variables or pointers.

Uninitialised Objects

Strictly speaking, "uninitialised objects" do not exist, as any object is (at least) initialised by the default constructor* of its class, but:

- If the default constructor is **automatically supplied** by the compiler, any data member of a built-in type receives the same initialisation as if it were not explicitly initialised.
- Same if a default constructor is **defined by the class itself** but forgoes to initialise some data members.

> The rules described so far apply recursively, i.e. what happens exactly depends on the type of the member: For basic types the rules from the previous page apply, for (nested) objects the rules on this page.

: The term default constructor might be a bit misleading in some cases, because it is easily confused with "(a) constructor supplied by default*". The latter **may or may not** be the case for the default constructor, but the technical meaning is this: "*the constructor that can be called without arguments*".

C++11 Brace Initialisation

Since C++11 braces are accepted in all initialising contexts shown so far (and some more). The equals sign is made optional.*

```
int x{42};
double pi{3.14152}
std::string greet{"hello"}
std::string line{80, '-'}
std::string empty{};
int primes[]{2, 3, 5, 7, 11};
Point origin{0.0, 0.0};
Point polygon[] {
    { 7.23, 12.1 },
    { 1, 3.5  },
    …
};
```

```
int x = {42};
double pi = {3.14152}
std::string greet = {"hello"}
std::string line = {80, '-'}
std::string empty = {};
int primes[] = {2, 3, 5, 7, 11};
Point origin = {0.0, 0.0};
Point polygon[] = {
    { 7.23, 12.1 },
    { 1, 3.5  },
    …
};
```

*: As no "C heritage" has to be taken into account the default constructor has an explicitly empty argument list, and initialising values that cannot be represented in the target type cause the compilation to fail, not just a warning.

C++11 Direct Member Initialisation

Class members can now be initialised directly (with their definition).

- The values specified are considered to be the default if there is **no member initialisation** via
  - the member initialisation list of the constructor used to initialise an object, or
  - as direct assignment in the construct block.
- Supported initialisation syntax is
  - classic assignment style,
  - brace initialisation, but
  - **not constructor style with round parenthesis!**

```
class Point {
    float x = 0.0;
    float y = 0.0;
    …
};
```

```
class Point {
    float x{0.0};
    float y{0.0};
    …
};
```

C++11: Compile-Time Initialisation (constexpr)

C++11 added the new keyword `constexpr` which may be applied to

- data definitions and
- function definitions

Compile-Time Initialised Data

By using `constexpr` with a data definition it is guaranteed that the variable can be initialised at compile time.

- The compiler checks for the initialiser whether its value can be determined, because it is
  - a literal constant,
  - an expressions, or
  - result of `constexpr` functions
- allowing to use such values in any context that requires compile time constants, most importantly
  - array dimensions
  - value template arguments
  - or as part of expressions or function call arguments when initialising some other `constexpr` data item.

> Compile time initialised data may also be stored in special ways, like in statically read-only or dynamically write-protected memory.

Compile-Time Callable Functions

By using `constexpr` with a function definition such function may (also) be called at compile time.

- In C++11 this was rather restricted:*
  - `constexpr` were limited to a single return statement
  - though recursive calls were allowed, and
  - recursion could be stopped with a conditional expression.
- In C++14 this is much relaxed, but there are still limitations, e.g.:*
  - **no** `asm`, `goto`, or `try-catch`,
  - variables only of literal types and with initialisation,
  - no `static` or `thread_local` storage duration.

> If a call to a `constexpr` function cannot be evaluated at compile time, a run time version is made available and accordingly called where the value it returns is needed.

*: For more information see: http://en.cppreference.com/w/cpp/language/constexpr

Constructors and Destructors

- General Purpose
- (No) Memory Allocation in Constructors
- Memory of Variable Size at Run-Time
- Compile Time vs. Run Time Sizing
- (No) Memory De-Allocation in Destructors
- Precautions in Memory Allocating Classes
- Exceptions in Constructors and Destructors
- Re-Using Constructors

General Purpose of Constructors and Destructors

Constructors and Destructors are one of the key additions of C++ over C.

Together they guarantee that

- initialisation will take place when an object comes into existence, and
- will be reliably reverted when an object ceases to exist.*

This is – of course – only true for objects with a scope-bound lifetime. In case of local `static` object

- the initialisation will take place only once (when the scope is entered the first time, and
- reverted after the main program has ended.

*: This easily extends to general resource allocation and deallocation, as is purposefully applied in the RAII-pattern.

(No) Memory Allocation in Constructors

A common misconception is that it is the constructor's responsibility to allocate memory for objects of its class:

Classic C++ initialisation:

```
class Point {
    float x;
    float y;
public:
    Point(float x_, float y_)
        : x(x_), y(y_)
    {}
};
```

C++11 brace initialisation:

```
class Point {
    float x;
    float y;
public:
    Point(float x_, float y_)
        : x{x_}, y{y_}
    {}
};
```

In this case there is already memory allocated when the constructor is called to initialise objects.

```
Point origin(0.0, 0.0);   // classic initialisation
Point other{3.1, 7.12};   // C++11 brace initialisation
Point zzz = {1.7, 0.0};   // as before, with equals sign
```

Memory of Variable Size at Run-Time

The reason for the misconception is that **some** classes (like `NamedPoint` below) will get space allocated for two `float`-s and one pointer, but the space pointed to by `name` is on the heap:

```
class NamedPoint {
    float x, y;
    char *name;
public:
    NamedPoint(const char * name_, float x_, float y_)
        : x{x_}, y{y_}
        , name{new char[std::strlen(name_)+1]} {
        std::strcpy(name, name_);
    }
};
```

[!] Therefore class `NamedPoint` needs a destructor – and probably a copy constructor and assignment too.*

*: Otherwise a resource leak or dangling pointers will almost certainly result as soon as objects of this class get created or assigned by copying from existing objects.

Compile Time vs. Run Time Sizing

The necessity to do memory allocation as part of the constructor – with the ultimate consequence to implemented more operations – does not arise if the space requirements are variable but fixed at compile time.*

```
class FString {
    const std::size_t N;
    char *data;
public:
    FString(std::size_t n)
        : N{n}
        , data{new char[N+1]} {
        data[0] = '\0';
    }
    ~FString() {
        delete[] data;
    }
};
…
FString fstr(20);
```

```
template<std::size_t N>
class FString {
    char data[N+1];
public:
    FString() {
        data[0] = '\0';
    }
};
…
FString<20> fstr;
```

- Above: sized at **compile time**.
- Left: sized at **run time**.

*: The design of a class for "fixed length" character strings taken as example is is only partially fleshed out here – in practice some more member functions would make sense.

(No) Memory De-Allocation in Destructors

Classes that do no (further) memory allocation in their constructor code also need no deallocation in their destructor – which may of course still be necessary for other reasons.

Continuing with the above examples, only `NamedPoint` has the necessity for a destructor:

```
class Point {
    …
    // no destructor required
    // to "de-allocate" memory
    ~Point() {
        // … but maybe for
        // other reasons
    }
    …
};
```

```
class NamedPoint {
    …
    ~NamedPoint() {
        delete[] name;
        // … maybe more to do
    }
    …
};
```

But: supplying a destructor which does nothing is neither necessary nor can it be considered "good style" since it may not be optimised-out and slow down the program each time an object ends it life.

Precautions in Memory Allocating Classes

Just copying the `data` pointer as the default constructor and assignment operator would do is surely the wrong thing, so these should at least be blocked,* which is particularly easy since C++11.

```
class NamedPoint {
    …
    NamedPoint(const NamedPoint &) =delete;
    NamedPoint& operator=(const NamedPoint &) =delete;
    …
};
```

---

*: Still better were to implememt the operations so that the `name` member gets properly cloned:

```
… // inside the class above
    NamedPoint(const NamedPoint &init)
        : x{init.x}, y{init.y}
        , name{std::strcpy(new char[std::strlen(init.name)+1], name)}
    {}
    NamedPoint& operator=(const NamedPoint &rhs) {
        if (this != &rhs) {
            x = rhs.x; y = rhs.y;
            delete[] name;
            name = std::strcpy(new char[std::strlen(rhs.name)+1], name);
        }
        return *this;
    }
```

Alternative Design with "Self-Contained" Helper Class

A still better design would put allocation and deallocation in a helper class which is self-contained with respect to memory management*

```
class FString {
    const std::size_t N;
    char *data;
public:
    FString(std::size_t n) : N{n}, data{new char[N+1]} {
        data[0] = '\0';
    }
    FString(const FString &init) : data{new char[init.N+1]} {
        std::strcpy(data, init.data);
    }
    ~FString() { delete[] data; }
    FString& operator=(const FString &rhs) {
        std::strncpy(data, rhs.data, N)[N] = '\0';
    }
    operator const char*() const { return data; }
};
```

*: Whether that way class `FString` is fleshed-out here is actually desirable for a generically usable "fixed length string class" is disputable, but it was chosen that way to mimic the prior approach (without helper) as much as possible.

Simplifying Clients with Self-Contained Members

Now class `NamedPoint` can be implemented without worrying about memory allocation and de-allocation:

```
class NamedPoint {
    float x, y;
    FString name;
public:
    Point(const char *name_, float x_, float y_)
        : x(x_), y(y_), name(std::strlen(name_)  {
        name = name_;
    }
    Point(const Point&) =default;            // if not =delete
    Point& operator=(const Point&) =default; // if not =delete
    …
};
```

The above also demonstrates the new syntax of C++11 to explicitly request the default copy constructor and assignment*, which means the operations are recursively forwarded to all of the members.

*: For backward compatibility, generating default versions of copy constructor and assignment will still happen in C++11 without the above.

Exceptions in Constructors and Destructors

Generally constructors and destructors

- may "fail" (in doing what they are expected to do) and
- (in general) **one** option is to inform clients about the problem by throwing an exception.

The topic is covered in more detail later, with exceptions, so the following pages only try to alert the reader for the problems caused by throwing from constructors and destructors.

Throwing from Constructors

When a constructor cannot establish the object state expected by the client (that created the object and caused the constructor to initialise it), the preferred way is to throw an exception.

Often this is the easiest way

- for the constructor code (that may just throw) ...
- ... and the client, that may catch the problem at an adequate location.

The only thing to remember here is that destructors are not enabled before the constructor ends normally,* hence partially created objects may required to

- wrap the constructor code into a `try`-block and
- handle exceptions in a local `catch`-block,
- finally re-throwing them (if the object is not left in a usable state).

*: This makes another good argument to use "self-contained" members, because then the compiler does the appropriate clean-up, as for all fully constructed member objects the destructor will be called when the construction of the containing object fails with an exception.

Throwing from Destructors

As – contrary to the constructor – the destructor usually just "cleans-up" and may not so easily run into resource limitations, there is much lesser reason for problems during destruction.

But the main problem is:

- Destructors are sometimes run during stack-unwinding after an exception has occurred.
- When a destructor then would throw (again), it is difficult to decide which exception should take priority.

Hence, when a destructor is run during stack-unwinding when handling an exception an some destructor it `throw`-s (again) the program is terminated.*

[!] Therefore the general rule is never to let an exception escape from a destructor.

*: A specific terminate handler for that case can be installed, but it may not (portably) return to its caller and continue.

Re-Using Constructors

C++11 has introduced two features to re-use code from existing constructors:

- Constructor Delegation and
- Constructor Inheritance

The former e.g. allows to have some few "*work-horse*" constructors that are called from a number of "*convenience constructors*".

The latter extends the possibility to "*import*" members from base classes with a `using`-directive into a derived class to constructors.*

*: It is not quite clear whether this was not allowed in C++98, maybe it was just an oversight ...

Constructor Delegation

With this feature – new in C++11 – a constructor of a class can be implemented in terms of (calls to) some other constructor:

- It works by calling the delegated-to constructor as part of the member-initialisation list of the delegating constructor.
- The delegating constructor must not have anything else besides the delegated-to constructor in its member initialisation list.
- Constructor delegation may be changed but must not be recursive.*

*: According to the C++11 standard a program with recursive constructor delegation is ill-formed and hence should not compile. But at least some older compilers seem not to detect this but produce a core dump when run.

Constructor Delegation Example

In the example following the `FString` constructor taking a `const char *` argument reuses the `FString` constructor taking an `std::size_t` argument for memory allocation:

```
class FString {
    const std::size_t N;
    const char *data;
public:
    FString(std::size_t n) : N{n}, data{new char[N+1]} {
        data[0] = '\0';
    }
    FString(const char *init) : FString(std::strlen(init)) {
 std::strcpy(data, init);
    }
    ~FString() {
        delete[] data;
    }
    …
};
```

Constructor Delegation and Exceptions

There is a small but useful change with respect to the rules when a destructor is enabled:

- This happens when the innermost (leaf-) constructor has reached its normal end.
- Therefore a constructor delegated-to should put the object
  - in a usable state – i.e. where all its invariants are established.
  - or at least provide a means by which the destructor can decide what has to be cleaned-up and what is not yet initialised at all.

[!] It should be noted that it is (somewhat) against the spirit of C++-constructors to leave objects in a "half-baked" state ...*

---

*: In general, if an object has constructors which are only used to delegate to and which do **not** establish all the invariants a user may expect, these constructors should probably not be `public` but `private` (or `protected` at most and properly documented with respect to the remaining initialisations.

Constructor Inheritance

If the base class constructors also work for the derived class, in C++11 it may be made visible in C++11 with a `using` directive:*

```
class Point {
protected:
    float x, y;
public:
    Point() : x(0.0), y(0.0) {}
    Point(float x_, float y_) : x(x_), y(y_) {}
    …
};
…
class MovablePoint : public Point {
public:
    using Point;
    void move(float dist_x, float dist_y) {
        x += dist_x;
        y += dist_y;
};
```

*: Typically a derived class adds its own data members and therefore requires extended initialisation, so it needs its own set of constructors.

Static Polymorphism

- Überladen von Funktionen
- Überladen von Operatoren
- Defaultwerte für Argumente
- Overloading and Overriding

Argument default values were included here as their applicability overlaps with function overloading, though – viewed from the hardware perspective – for non-`inline` functions they are implemented quite differently.

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 Operator-Gruppen 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 built-in-types cannot be changed. If you want to come close to the behaviour of a given built-in 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 Rückgabe-Typ ist dabei beliebig, die `return`-Anweisung muss natürlich vom Typ her passend sein.

Genauer gesagt, der Ausdruck hinter `return` muss

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

*: See also here for an interresting short-hand notation possible with C++11 brace initialisers.

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 Rückgabe-Typ 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 persistent" (like a reference), copy assignment would be used instead, though a move could be enforced then: `c = std::move( … );`

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 desperate because of an completely unexplainable syntax error in your source code.

Overloading and Overriding

The above two terms are sometimes confused, though – if used precisely – they have a well defined, different meaning:*

- *Overloading* means there are two (or more) functions with identically spelled function names but with different parameter lists so that the compiler can select one based on the actual call arguments.
- *Overriding* means that some class uses the same name for a member function that has already been used in one of its base classes.

As the usual translation for "overriding" to German is "überschreiben", by sloppy back-translation sometimes the term "overwriting" has been introduced as a synonym for "overriding".

*: Disregarding the fact that an object in a member function call is syntactically not part of the argument list (but goes to the left, separated with a dot), from a purely technical perspective the object involved in the call is just another argument and hence "under the hood" overloading and overriding are actually related.

Overriding Example (1)

In the following example there are three overloaded member functions `foo`:

```
class Base {
    …
    void foo(int);
    void foo(double);
    void foo(const char *);
    …
};
```

It are **all(!)** of the above that get overridden by a **single** member function `foo` in a derived class:

```
class Derived : public Base {
    …
    void foo(double);
    …
};
```

Overriding Example (2)

How the following calls are resolved sometimes even surprises seasoned C++ developers:

```
Base b;
b.foo(42);        // calls Base::foo(int) -- no surprise
b.foo(3.14);      // calls Base::foo(double) -- no surprise
b.foo("hello");   // calls Base::foo(const char *) -- no surprise
…
Derived d;
d.foo(3.14);      // calls overriding Derived::foo(double)
d.foo(42);        // as before(!) with type conversion of argument
d.foo("hello");   // ERROR (does not compile)
```

[!] Especially if there are automatically applied type conversions (like above from `int` to `double`) the problem might go unnoticed for a long time.

Overriding and the `using` Directive

If a derived class shall inherit **all** overloads of its base class and then **some** are selectively overridden, the `using` directive will help:

```
class Derived : public Base {
    …
    using Base::foo;
    void foo(double);
    …
};
…
Derived d;
d.foo(3.14);     // calls overriding Derived::foo(double)
d.foo(42);       // calls Base::foo(int) -- no surprise
d.foo("hello");  // calls Base::foo(const char *) -- no surprise
```

[!] There might still be surprises in case of argument types that do not exactly match and hence undergo a conversion.

Luckily many oversights falling into that category of problems will manifest themselves in ambiguities and cause the compilation to fail.

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 *`).

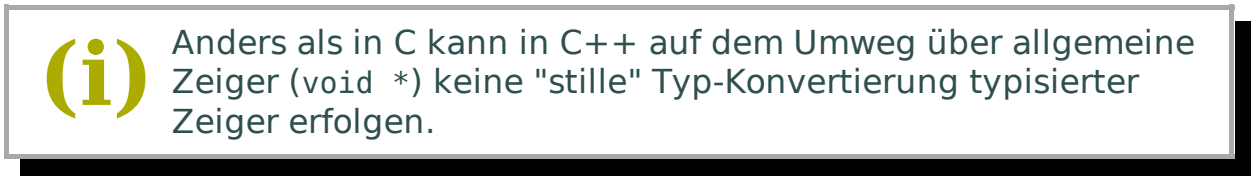
Inwieweit dies auch für die anderen, oben aufgezählten Fälle gilt, hängt von Hardware-Details ab, insbesondere der Repräsentation von Aufzählungstypen und Zeigern.

*: 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**.

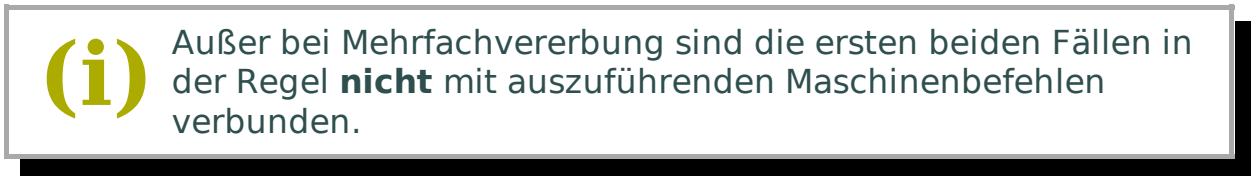
Mit dem letzten Punkt wurde die Typ-Sicherheit von C++ gegenüber C an einer kritischen Stelle verbessert.*



*: 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 substitutability 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 (Down-Cast)
  - automatisch nur abgeleitete Klassen → Basisklassen (Up-Cast)

---

*: 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 Down-Casts 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 Programmierung 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

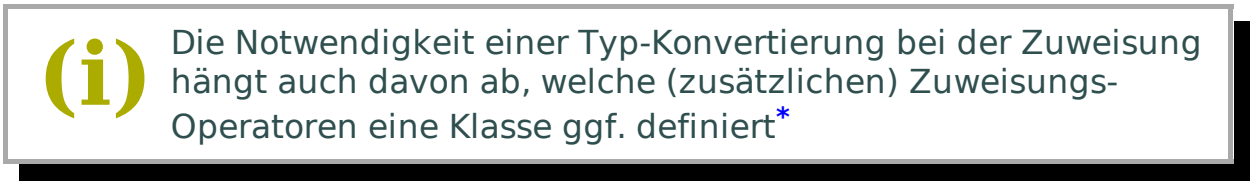
- genau ein Argument besitzen und
- **nicht** mit dem Schlüsselwort `explicit` markiert sind.

```
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 would 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
```



*: An assignment operator taking **exactly** the type of the expression of its right hand operand as argument will always be preferred (and of course applied as there is no need for a conversion). In case the assignment operator takes a `const MyClass&` argument a temporary will be created if a non-`explicit` c'tor or a type-cast operator is available as automatic conversion.

Brace Initialisers as Short-Hand Notation

In some contexts brace initialisation syntax may also be used as short-hand notation for constructor calls, without naming the class:

```
class MyClass {
   …
   MyClass();
   MyClass(int);
   MyClass(const char*, double);
   …
};
…
void foo(MyClass);
foo({});                // foo(C{});
foo({42});              // foo(C{42});
…
MyClass bar() {
   …
   return {"hi", 3.14};   // return C{"hi", 3.14}
}
```

The only necessary precondition is that the constructors to be applied are not `explicit`, their argument count does not matter.

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 folgenden 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.