C++ FOR
(Monday Afternoon)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

More C++11/14 Extensions

- Extensions to Literals
- Static Assertions
- Explicit `nullptr` Value and Type
- Initialiser Lists
- `auto`-typed Variables (and Objects)
- Trailing Return-Type Syntax
- Range (-based) `for`-Loops
- Alternative for Type Definitions

Extensions to Writing Literals

C++11 extended the syntax for literals in various ways:

- Character and character string literals:
  - UTF-8 and UTF-16 encoding can be requested
  - Quoting of special characters can be reduced
- Numeric literals:
  - Readability can be improved by structuring
  - Notation can be in binary
- All kinds of literals:
  - May take user defined suffixes

Specifying Encoding of Characters

Since C++11 characters and string literals can explicitly specify:

- UTF-8 encoding by the prefix `u8` (8 bit code units)
- UTF-16 encoding with the prefix `u` (16 bit code units).
- UTF-32 encoding with the prefix `U` (32 bit code units).

Note that the `L`-prefix from C++98 is implementation defined with respect to code unit size (typically 16 bit on MS-Windows and 32 bit on Linux and most other execution environments derived from Unix) and encoding.

*: If the full code point space of UCS (with the uppermost limit at 0x10FFFF as set by ISO/IEC10646) is used, a single code point may take up to four code units in UTF-8 and one or two code units in UTF-16. With [UTF-32] there is always a 1:1 mapping from between code points and code units – but not necessarily between code points and user perceived characters! Only when the character set is limited to UCS-2 – which contains all of the basic multilingual plane (BMP) – there is a 1:1 mapping from code points to code units in UTF-16 too. (But again: code points are not necessarily user perceived characters!) As there is currently **very limited support** in standard C++ for conversions between encodings and other common operations on multilingual text, usually the (free) ICU-Library is recommended for text-centric applications with serious needs for internationalisation and appropriate localisation.

Raw String Literals

The requirement to quote a number of special characters in string literals is alleviated in C++11 with the introduction of raw string literals:

```
constexpr char text[] = R"(
… any content … no necessity for quoting here …
… no special interpretation of certain characters …
)"; // <-- except this end of raw string indicator
```

Also the end marker of raw string literals can be freely chosen and the prefix may be combined with an UTF encoding specifier:

```
… u8R"!!end-of-text!!( … whatever … )!!end-of-text!!" …
```

The usual splicing of lines ending in a backslash will **not** happen in raw string literals as the first and second of the Phases of Translation do not apply to their content.

For more information on raw string literals see:
http://en.cppreference.com/w/cpp/language/string_literal

Structuring Numeric Literals for Readability

Since C++14 it is possible to write apostrophes inside numeric literals:

```
int x = 250'000;        // 250-thousand (250000)
int y = 12'500'000;     // 12-million 500-thousand (12500000)
int z = 1'500'000'000; // one and a half (US-) billion (1500000000)
                        // or: 1,5 Milliarden in German
auto t = 0.125'379'12; // if t is a time in seconds this would add
                        // up to: 125 milliseconds
                        //      + 379 microseconds
                        //      + 120 nanosecond (yes, 120!)
```

Note that apostrophes may be inserted anywhere in a numeric literal, not just to separate groups of three or groups of the same length.

*: It can be expected that this feature will confuse a number of editors or C++ IDEs with syntax-highlighting – until such are adapted. (This is possible visible on this page, which uses a JavaScript-plugin for syntax highlighting, that is not yet adapted to that feature at the time of writing this page.)

Numeric Literals in Binary Notation

C++14 introduced a syntax for binary literals, starting with `0b`:

```
… 0b101010 …                 // same as: 052, or 42, or 0x2A
… 0b'00'1101'101'110101' …   // same as: 0x1B75
```

Interspersing apostrophs anywhere, like in the second example, might be used to reflect a domain specific grouping of bits (2-4-3-6 in the example).

See also:
http://en.cppreference.com/w/cpp/language/integer_literal

User Defined Literals

C++11 introduced a new form of operator overloading to allow user specific suffix to numeric, character and string literals.

There are many possible uses, one is to provide readable, strongly typed units for literals representing physical quantities.

C++14 applied that feature to provide a number of additional suffixes, e.g.*

- `"hello"s` – represents an object of type `const std::string` initialised with `hello`.
- `2.0i` – represents an object of type `std::complex` with the real part set to zero and the imaginary part set to two.

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

User Defined Literals Example (1)

As a motivating example assume a class representing a length as a distinct type, i.e. one that should not mix with ordinary (dimensionless) numbers.

The initial shot could look as follows:

```
class Length {
    long double metres;
    constexpr Length(long double m) : metres(m) {}
    …
};
```

Internally a `long double` is used to represents some length in metres, but to avoid wrong assumptions on the user's side the constructor is private.

User Defined Literals Example (2)

To create objects of this class it provides a number of helper functions:

```
class Length {
    …
public:
    static constexpr Length m(long double v) {
        return Length(v);
    }
    static constexpr Length km(long double v) {
        return Length(v*1000.0);
    }
    static constexpr Length mm(long double v) {
        return Length(v/1000.0);
    }
};
```

User Defined Literals Example (3)

If the class also provides some output operation, like

```
#include <iostream>
…
class Length {
    …
    friend
    std::ostream &
    operator<<(std::ostream &lhs, const Length &rhs) {
        return lhs << rhs.metres << "m";
    }
}
```

the following small demo program should compile:

```
int main() {
    auto dtm = Length::km(385000);
    auto inch = Length::mm(25.4);
    std::cout << "Distance to moon is " << dtm << std::endl;
    std::cout << "One inch is " << inch << std::endl;
}
```

User Defined Literals Example (4)

User defined literals would provide the following convenience:

```
auto dtm = 385000_km;
auto inch = 25.4_mm;
```

In the following implementation they delegate to a single helper

```
constexpr Length operator"" _km(unsigned long long v) {
    return Length::m(v*1000.0);
}
constexpr Length operator"" _mm(long double v) {
    return Length::m(v/1000.0);
}
```

so the other helpers are not any more necessary – even no helper at all were necessary if the functions were turned into friends of `Length`.

Static Assertions

C++11 introduced the compile time directive `static_assert` to allow for tests that may terminate compilation, if some condition is not fulfilled.

C++1y will make the second argument of `static_assert` optional.

Static Assertion Example

Application for static assertions are manifold.

The following example shows how a user defined literal could check for a certain properties of the value supplied, e.g. when only even numbers from the range `1` to `1001` make sense:

```
MyClass operator"" _mysfx(unsigned long long v) {
    static_assert(1 <= v && v <= 1001, "value out of range");
    static_assert((v % 2) == 0, "only even numbers accepted");
    return {v}; // assuming MyClass has a non-explicit constructor
                // taking an argument of type unsigned long long
}
…
… 12_mysfx …     // OK
… 123_mysfx …    // ERROR (due to check with static assert)
… 12.3_mysfx …   // ERROR (because operator"" _mysfx()
                 //        expects an integral value)
```

Static Assertion Recommendations

- If some erroneous condition can be checked at compile time, this is **always** preferable to a test at run time.
- Static assertions require expressions that can be evaluated at compile time, hence `constexpr` helper functions will often come in handy.
- To unlock the full potential of static assertions, be sure to get a good understanding of what is provided via the header file `<type_traits>`.

Explicit `nullptr` Value and Type

C++11 introduced the new keyword `nullptr` of type `std::nullptr_t`*

- to denote a literal constant for an address,
- which will never be given to a valid
- variable of some built-in type or
- object of some class.

Before, either the preprocessor macro `NULL` or a literal `0` had to be used.

*: Other as for `void *`, **into which** any other pointer type will be converted if necessary, the type `std::nullptr_t` which converts vice versa converts **into** any other pointer type, but its only possible value is `nullptr`.

`nullptr` Example (1)

The use of `nullptr` as an argument may help to resolve overloads (and avoid surprises) in some cases like the following.

Assume a function with overloads for integral values and char pointers:

```
…
void foo(int);
void foo(const char *);
…
```

What is called here?

```
foo(NULL);
```

Here the intent is clear:

```
foo(0);
foo(nullptr);
```

Since `nullptr` is a keyword, it is available without any include file, but to use `std::nullptr_t` the header `<cstddef>` has to be included.

`nullptr` Example (2)

Even `nullptr` is not helpful if there are several overloads on pointers:

```
…
void foo(void *);
void foo(const char *);
void foo(int *);
void foo(MyClass *);
…
```

Given a number of overloads for different kinds of pointers as shown left, the following call is ambiguous:

```
foo(nullptr);   // ambiguous
```

The solution is to cast the `nullptr` (or `0` or `NULL`) to the correct type:*

```
foo(static_cast<int*>(nullptr)); // OK (in C++11, as nullptr used)
foo(static_cast<int*>(0));       // OK (in C++98 and C++11)
foo((int*)0);   // also OK (but C-style cast is deprecated in C++)
```

*: If all the pointer overloads test for the `nullptr`-case – like it should be expected for a wide interface in which the client may legally hand-over that value – and if all react in the same way, an alternative were to add another overload for `std__nullptr_t`

Usage Recommendations for `nullptr`

- With `nullptr` there is little reason to stay with the classic alternatives.*
- Any use `0` or `NULL` in a pointer context can safely be changed to `nullptr`.
- Care has to be taken when `0` or `NULL` is explicitly casted to some pointer type.
- Some (but not all) of such usages will still require the use of a cast.

[!] A `reinterpret_cast` on a `nullptr` (or `0` or `NULL`) should always be looked-at with suspicion, as it just reuses the `bits and bytes` given to it with a different interpretation.

*: One exception were backward-compatibility to compilers not (yet) implementing C++11.

Initialiser Lists

As new type the template `std::initializer_list` has been introduced with C++11.

- Initializer lists may be used as argument to constructors to fill containers.
- The type of the contained initialisers may be
  - deduced from the call context and accordingly converted (if mapped to an `std::initializer_list<T>`-s with a known `T`), or
  - must at least be unique (if mapped to an `std::initializer_list<T>` with a dependant type `T`).

Initialiser List Example

A (hypothetical) class `Polygon` might be initialised as follows:*

```
#include <initializer_list>
…
class Polygon {
    std::vector<Point> points;
public:
    Polygon(std::initializer_list<Point> init) {
        std::copy(init.begin(), init.end(),
                  std::back_inserter(points));
   }
   …
};
…
Polygon drawing{ {Point{7, 12}, Point{3, 8}, Point{1, 5}} };
```

---

*: Also the classic initialisation syntax may be used as long as the initialiser list is enclosed in curly braces. (Also, as shown in the last line, the class name `Point` may be omitted, but only with brace initialisers.)

```
// alternatives to the above:
Polygon drawing = {Point{7, 12}, Point{3, 8}, Point{1, 5}};
Polygon drawing({Point{7, 12}, Point{3, 8}, Point{1, 5}});
Polygon drawing({Point(7, 12), Point(3, 8), Point(1, 5)});
Polygon drawing({ {7, 12}, {3, 8}, {1, 5} });
```

Initialiser List Recommendations

- Initialiser lists allow more compact initialisation of container-like classes.
- Though they are not limited to that area, they might e.g. also be used with range-`for`.
- Aside from simple uses, as for (by-) value elements with copy initialisation, be aware of some limitations, like e.g.:*
  - Initialiser lists containing move-only types can still be handed over as function arguments via rvalue references …
  - … but elements from such lists cannot be move-out, only accessed in place.

*: Also if some such limitations may not be obvious at the first glance, they have a natural reason that becomes evident if alternative choices are considered.

`auto`-typed Variables and Objects

Instead of supplying the type in a definition of a variable or object, C++11 allows the use of the keyword `auto`.*

- There must be an initialising expression.
- The type of the definition is taken from that expression.
- If `auto` is used unadorned, then the type is the type of the initialising expression with `const`, `volatile` and references (`&`, `&&`) stripped away.

The type deduction rules for `auto` will be considered in more detail later.

*: Actually this is a change of the meaning `auto` once had in C, which is to request a stack-based variable with the option to omit the type and use the default `int`. As types were not any more optional in C++ since long, the new meaning cannot cause silent changes.

Examples for `auto`-typed Variables (and Objects)

Though the main use cases for `auto` as type of a variable or object comes in later examples (with more complex types), here are some trivial cases using `auto` instead of an explicit type, which behave "like expected".

Given the following definitions ...

```
int a = 0;
long int b = 42;
const int &r = a;
const std::string s("hi");
```

... `auto` may be used as shown:

```
// type of variable is:
auto u = 42LL;    /*1*/
auto v = 2*b;     /*2*/
auto w = r;       /*3*/
auto x = s;       /*4*/
auto y = "hi!";   /*5*/
auto z = nullptr; /*6*/
```

Types were not named on the right, in case you want try yourself :-) ...*

*: ... here they are, in the order of appearance:

- `u` has type `long long`
- `v` has type `long`
- `w` has type `int` (plain `int`, no `const`, no reference)
- `x` has type `std::string` (no `const`)
- `y` has type `const char *`
- `z` has type `std::nullptr_t`

Usage Recommendations for `auto`-typed Variables

- Advantages for simple types (like shown so far) are debatable.
- Some C++-Gurus nevertheless recommend to **always** prefer `auto`.*
- One obvious advantage is that it guarantees initialisation.
- If a specific type is required, it may also "go to the right" like in:
  ```
  auto x = std::uint16_t{0xFFFF};
  ```
- This makes code look more unique as the above is similar to e.g.:
  ```
  auto p = new MyClass{"hi!", 3.14};
  ```

[!] Do not blindly change any use of typed variables to plain `auto`, as not everybody may be familiar with how to specify types in literal initialisers. Especially – for the moment – do not combine `auto` with brace initialisers.

*: Here is a section from a video in which Herb Sutter gives his arguments why he would prefer to see a change of customs ... but also concedes to other experts to have "the right to have a different opinion": https://www.youtube.com/watch?feature=player_detailpage&v=xnqTKD8uD64#t=1704

Intermezzo: `auto` and Brace Initialisation Corner Cases

Minimised Number of Rules

The fewer the rules exist, the lesser is to learn and to remember. All cases, from trivial to highly complicated, can be explained with the given set, though the outcome is (sometimes) not the most convenient.

Principle of Least Surprise

If a feature "just works" in a way that meets the expectations of "most users", then nothing at all is to be learned … except – maybe – for some dark corner cases, where it works not as expected.

For new features and their cooperation with each other the trade-off is sometimes hard to make and an initial try may come out wrong:

All below is currently deduced as `std::initializer_list`:

```
auto x{2, 3, 5};
auto x = {2, 3, 5};
auto y{42};
auto y = {42};
```

Expected for C++1y and already implemented in some compilers:*

```
auto x{2, 3, 5}; // illegal
auto x = {2, 3, 5}; // list
auto y{42}; // deduced as int
auto y = {42}; // list
```

*: Surely this does not minimise the number of rules … so does it at least avoid surprises?

Recommendations for `auto` and Brace Initialisation

The least to state is that brace initialisation in cooperation with `auto` did probably not "simplify" learning C++, even if no legacy code were to be maintained and all the "old" initialisation syntax could be forgotten.

(i) For more information, especially when an why the rules do not produce the expected behavior and therefore will probably change with C++1y (and are already implemented in the new way in MS-VC++ and the upcoming releases GCC 5.0) see: http://arne-mertz.de/2015/02/type-deduction-and-braced-initializers/

For now the most easy to follow recommendation to avoid surprises in case of brace initialisation and `auto`-typed variables is this:

- Do not combine `auto`-typed variables with brace initialisation.

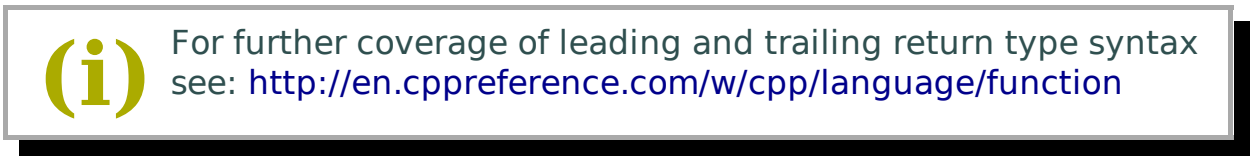
Trailing Return-Type Syntax

Since C++11 `auto` can also be used to specify the return type of a function after its formal argument list, like in:

```
const char *foo() {
    …
    return "hello, world";
}
```

```
auto foo( … ) -> const char * {
    …
    return "hello, world";
}
```

In these simple examples an advantage is hard to see, but they will become visible in later (more advanced) use cases.



Completely Omitted Return-Type

Since C++14 when replacing the leading return type of a function with `auto`, the trailing return type may also be omitted in some cases.

If a function is only declared but not defined, the return type must be given:

```
auto foo() -> const char *; // trailing return type required
```

Otherwise the return type is deduced if there is only a single return statement or all return statements have the same type:

```
auto foo()
    …
    return "hello, world";
}
```

More than one return statement and different types:

```
auto foo() -> const char * {
    if ( … )
        return nullptr;
    …
    return "hello, world";
}
```

Usage Recommendations for Trailing Return Types

- In many (simple) cases using the trailing return type syntax has no direct benefit.
- Instead it requires more typing and breaks with the familiar "look & feel" of function definitions.
- Nevertheless deciding to **require** the use of trailing return types – e.g. per local *Style Guide* – might help to produce a more consistent programming style ... (on the long run, after most traditional usages are changed).
- Also be prepared for rejection by a substantial fraction of developers, clinging to traditional style (with various arguments).

[!] Do **not omit** trailing return types completely unless you surely know you will **never** have to compile in environments less recent than C++14.

Range (-based) `for`-Loops

C++11 introduced a unified syntax to loop over all elements of a collection.

It looks similar to the classic `for`-Syntax, but inside the parentheses following the keyword `for`

- a place-holder variable
- is separated by a colon (`:`)
- from a collection.

Its use becomes usually obvious from some characteristic examples, like those shown in the next pages, and also feels quite natural soon.

Example: Range-For over Classic Array

Using range-`for` to read or modifying all elements in a classic array …

```
int data[100];
… // fill with 100 values
```

… the code on the right side is equivalent to the code below, accessing all elements of data one after the other by index …

```
for (int i = 0; i < 100; ++i) {
    int e = data[i];
    … // read data[i] via e
}
…
for (int i = 0; i < 100; ++i) {
    int &v = data[i];
    … // modify data[i] via v
}
```

```
for (int e : data)
    … // sequentially read
      // values from data via e
…
for (int &v : data)
    … // sequentially modify
      // values from data via v
```

… or – more C-style – via pointer:*

```
for (int *p = data;
    p < data+100; ++p)
    … // read data via *p
…
for (int *p = data;
    p < data+100; ++p)
    … // modify data via *p
```

*: Performance evaluations typically show neither version has an advantage over the other.

Example: Range-For over STL-Vector

One of the main advantages of range-`for` is its uniform syntax that applies to STL containers too (below left) instead of classic iterator loop (right):

```
std::vector<int> data;
… // fill with values
for (int e : data)
    … // read data values
      // via e
…
for (int &v : data)
    … // modify data values
      // via v
```

```
typedef
vector<int>::iterator Iter;
for (Iter it = data.begin();
          it != data.end();
        ++it)
    … // access (read or
      // modify) data
      // values via *it
```

```
for (auto e : data)
   … // access via copy
for (const auto &e : data)
   … // access via reference
```

Also `auto` comes in handy as well for read-only (left) as for modifying access (below):*

```
for (auto &e : data)
   … // access via reference
```

*: Actually this will allow for modifying access only if `data´ is not`const-qualified, because otherwise the place-holder of the range-`for`loop would be deduced as`const` reference!

Example: Range-For over STL-Map

In case of an STL-Map the place-holder is in the range-`for` loop is a pair of the maps key value-type … which might seem inconvenient to specifiy …

```
std::map<std::string, int> data:
… // fill with key-value pairs
for (std::pair<std::string, int> e : data)
    … // access key via e.first and
      // associated data via e.second
```

… but again `auto` comes in handy:

```
for (const auto &e : data)
    … // access key via e.first
      // and associated data
      // read-only via e.second
```

```
for (auto &e : data)
    … // access key via e.first
      // and associated data
      // modifiable via e.second
```

Note that the key is always non-modifiable!

Example: Range-For over Non-Standard Containers

It is well possible to use range-`for` loops with non-standard containers

```
MyContainer data;
… // fill data with values
for (auto e : data) … // read (by copy) via e
for (const auto &e : data) … // read (efficiently) via e
for (auto &e : data) … // access modifiable via e
```

given one of the following helpers exist:*

Either:
`MyContainer` provides (the standard STL container interface with) the **member functions**:

- `… MyContainer::begin() …`
- `… MyContainer::end() ….`

Or:
There are overloads with an argument of type `MyContainer` for the **global functions**:

- `… begin(… MyContainer& …) …`
- `… end(… MyContainer& …) ….`

*: In both cases the return type and some details of the argument transfer in case of global functions are left unspecified here. But what is returned from these functions must be equality-comparable (and eventually compare unequal if the loop is expected terminate regularly) and there must be increment and dereference operations for the returned type.

Example: Range-For with Initialiser List

It is also possible to combine range-`for` with `std::initializer_list`-s:

```
enum class Color { Red, Blue, Green, Unspecified = -1 };
…
for (auto c : { Color::Red, Color::Blue, Color::Green }) {
    … // do something with c
}
```

If such lists (of all values of some enumeration type) are required in many places, they may be specified as initialised constants, preferably close to the definition of the enumeration, so that both can be easily maintained in parallel.

```
enum class Color { Red, Blue, Green, Unspecified = -1 };
constexpr auto ALL_COLORS = { Color::Red, Color::Blue, Color::Green };
…
for (auto c : ALL_COLORS ) …
```

Usage Recommendations for Range-`for` Loops

- Replacing typical iterator loops over containers with range-`for` usually causes no problems and will result in code that is more readable and easier to maintain.
- The same is often true for `std::for_each`, at least as long as a container is processed **completely**.
- For processing container sub-ranges `std::for_each` is still useful.
- Non-standard containers should **strongly consider** to provide the interface required by range-`for` to iterate over their content.

[!] Efficiency problems may result if some container holds large objects which are not cheap to copy and the place-holder in a range-`for` loop is not specified as reference.*

*: The recommendation for *generic code* is to use `auto&&` because this will always give optimal results. On the other hand, `auto&&` is a language construct which should not be carelessly (i.e. without really understanding the implications).

Alternative for Type Definitions

There is a completely new way to specify a type alias in C++11.

Its syntax is*

- the keyword `using` followed by
- the new type name
- an equals sign
- some existing type.

The whole construct may also be specified as a template, requiring (one or more) instantiation types when used.

*: Since long `typedef` is a feature of C and hence became part of C++ too. Though, looking closely, its syntax seems not straight-forward compared to the usual assignment syntax, as the new "name" that gets an existing type as "value" is on the right.

New C++11 Type Aliases Example (1)

How to use the feature becomes quickly obvious from a few examples:*

```
using counter_type = int;
using MyDataContainerType = std::vector<int>;
using named_counter_t = std::map<std::string, counter_type>;
```

The new names are aliases (only).

Especially they do not constitute new types on which overloading were possible, i.e. the semantics are the same as for `typedef`.

New C++11 Type Aliases Example (2)

Often local type definitions or aliases may help to make code more readable.

The `typedef` may look more familiar as everybody knows it ...

```
typedef map<std::string, unsigned long>::iterator MapIterator;
for (MapIterator it = data.begin(); it != data.end(); ++it)
    …
```

... but aliases with `using` may appears more natural, as the new identifier being defined goes to the left and what it stands for to the right.*

```
using MapIterator = map<std::string, unsigned long>::iterator;
for (MapIterator it = data.begin(); it != data.end(); ++it)
    …

}
```

*: It is probably because the switched around positions of the new type name and what it is aliased to, together with the equals sign that makes it easy see the role of each part, what makes the new syntax with `using` better readable.

New C++11 Type Aliases Example (3)

A particularly convincing example for the improved readability is this:

- A pointer to a function
- taking an argument list of
  - a pointer to non-modifiable characters and
  - an integer
- returning a result of type `bool`.

Usual C style (up to C++98):

```
typedef bool (*MyFuncPtr)(const char *, int);
```

With C++11 type aliases:

```
using MyFuncPtr = bool (*)(const char *, int);
```

New C++11 Type Aliases Example (3)

An additional advantage of the new syntax is that it allows to be templated:*

```
template<typename CounterType>
using NamedCounters = std::map<std::string, CounterType>;
```

It also allows for default arguments, but an empty angle bracket remains if the default is to be used (and no other template arguments remain).

```
template<typename CounterType = long long>
using NamedCounters = std::map<std::string, CounterType>;
…
NamedCounters<> counters;
```

---

*: By using the new type aliases in C++14 (in addition to) the classic style in which Type Traits were made available in C++11, a lot of (ugly and seemingly redundant) `typename ...::type` constructs can be simplified, as the following are equivalent:

- `typename std::enable_if<(N > 0), typename std::add_lvalue_reference<T>::type>::type ...`
- `std::enable_if_t<(N > 0), std::add_reference_t<T>> ...`

Usage Recommendations for C++11 Type Aliases

- As type aliases in C++11 style are much more readable there is little reason not to use them.
- As they have block scope introducing an alias with an expressive name can make (closely following) code easier understandable.
- Templated type aliases allow elegant solutions for a number of problems that formerly required more effort, like derived classes with a certain amount of duplicated code.
- Beyond the definition syntax nothing new needs to be learned since the semantics of C++11 type aliases are identical to `typedef`-s.

[!] Identical semantics also means that C++11 type aliases do not constitute types of their own, so **overloading will not work** for different aliases mapping to the same underlying type.

Basics of Type Deduction

- Type Deduction – Why and Where?
- Type Deduction for `auto` Variables
- Type Deduction for Templates
- More Type Deduction Scenarios

These chapter centrally introduces into type deduction so that this topic is not sprinkled through the other parts, where type deduction occurs.

Type Deduction ... Why and Where

There are several places where type deduction is done by the compiler. Some have to do with convenience, but most have to do with Generic Programming, which is the C++ solution to the following observation:

- Some code, especially reusable code designated for a library, only differs in types, not in algorithm.
- Code duplication can not be the solution, as it violates the DRY-Principle.
- To weave together generic code and the places of its use with specific types, the compiler needs to carry out type deductions.

> The latter may seem – and are – trivial in simple cases but can also become quite complicated, because the underlying rules try too guarantee an outcome according to *what the users (of generic code) expect (in their special use case*.

Type Deduction for `auto`-Variables

Using `auto` as type was introduced into C++11 mainly as a convenience feature*, though in the meantime using `auto` on a regular base becomes part of some style recommendations.

The basic rules, how the type is deduced from the initialising expression, differs between the unadorned use and possible declarators like `*`, `&`, `&&` (pointer, reference) and qualifiers like `const` or `volatile`, e.g.:

```
auto x = …          // any initialising expression
auto *p = …         // initialiser must denote an address
const auto &r = … // initialiser must be an lvalue but
                    // access does not allow modifications
```

Type Deduction unadorned `auto`

The rules follow the prevailing user expectations* that `const` will not get part of the type of the defined variable

```
auto x = 42; // x is type int (NOT const int)
auto y = 0L; // y is type long (NOT const long)
```

despite the fact that `42` and `0L` are both literals and hence constant.

This logically extends to `const` qualified names and references used as initialising expressions

```
const unsigned int MAX = 1000;
…
auto limit = MAX; // limit is type unsigned int (NOT const)
```

where the `const` qualifier does **not** propagate from `MAX` as initialiser into the type deduced for `auto`.

---

*: Alternatively the reasoning about why these rules make sense can be built on the argument that the alternative – carrying over `const`-ness into the type deduced by `auto` – would make that feature much less useful: you may much more often need variables initialised with a value from a constant, as you need (another) name for an existing constant.

Type Deduction `auto` and References

With respect to references there is a difference between

- an initialising expression of reference type, and
- a reference declarator as part of the `auto` type.

```
int s;
int &r;
…
auto x = r;   // x is int (NOT int reference)
auto& y = s;  // y is reference to int (and s needs to be an lvalue)
auto& z = r;  // z is reference to int (and r needs to be an lvalue)
```

Again the rules meet the user expectations, especially for `y` and `z` which of course need to be references as to expect from the involved declarator (`&`).

Furthermore, in case of `auto&` it makes sense to carry over `const`-ness into the deduced type:

```
const int s2 = -1;
…
auto& z2 = s2; // s2 is reference to const int
```

Type Deduction for `auto` with Adornments

If a variable type is to be deduced automatically and it is not plain `auto`

- the adornments to `auto` get part of the deduced type,
- the type of the initialising expression may be limited accordingly, and
- `const` qualifiers from the initialiser become part of the deduced type.

Given existing variables (below) used to initialise `auto` typed variables (right):

```
int a;
const int b = 20;
```

```
const auto v = a;     /*1*/
auto *p1 = &a;        /*2*/
auto *p2 = &b;        /*3*/
const auto *p3 = &a;  /*4*/
auto &r1 = a;         /*5*/
auto &r2 = b;         /*6*/
const auto &r3 = a;   /*7*/
```

If you want to try yourself … *

*: … the result is:

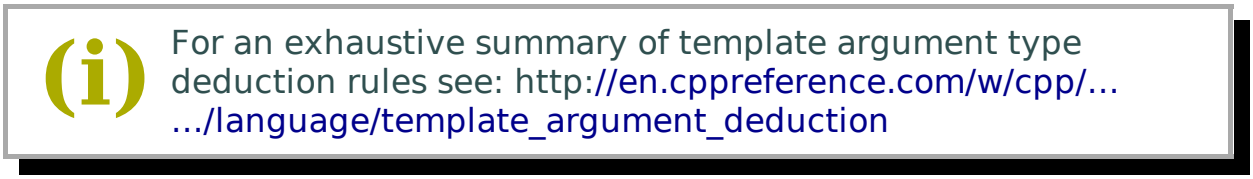
`/*1*/` is `const int`
`/*2*/` is pointer to `int`
`/*3*/` is pointer to `const int`
`/*4*/` is pointer to `const int` (even though `a` is not `const`)
`/*5*/` is reference to `int`
`/*6*/` is reference to `const int`
`/*7*/` is reference to `const int` (even though `a` is not `const`)

Type Deduction for Templates

Before `auto` was introduced to derived the type of a variable from the initialising expression with C++11, there was already a set of type deduction rules in C++98, used in case of template function parameters:*

```
template<typename T> void foo(T arg) { … }
template<typename T> void bar(T &ref) { … }
template<typename T> void bar(const T *ptr) { … }
```

In this case a type were to be deduced from the call argument.



*: In fact, as templates are much older, their type deduction rules were used to guided the rules for `auto`, but there are although subtle differences.

Template Type Deduction Example (1)

Assuming the template functions from the previous page and given existing variables (as below) valid calls might look like on the right:

```
int a;
const int b = 20;
```

```
foo(12);  /*1*/
foo(a);   /*2*/
foo(b);   /*3*/
bar(a);   /*4*/
bar(b);   /*5*/
baz(&a);  /*6*/
baz(&b);  /*7*/
```

The key point to understand here is that actually two types are deduced:

- The type formally represented by `T` and
- the type of the argument (`arg`, `ref`, or `ptr`).

Again, if you want to try yourself … *

---

*: … the result is:

`/*1*/` to `/*3*/` `T` and `arg` is type `int` (call by value)
`/*4*/` `T` is type `int` and `ref` is type reference for `int`
`/*5*/` `T` is type `const int` and `ref` is type reference for `int`
`/*6*/` `T` is type `int` and `ptr` is type pointer to `const int`
`/*7*/` `T` is type `int` and `ptr` is type pointer to `const int`

Make sure you did not overlooked the different use of `const` in the definitions of `bar` and `baz`!

Template Type Deduction Example (2)

In case it is not obvious so far, template typed deduction may look "deeply" into an argument definition to find the templated type and hence will also manage deduce T in the following examples:

```
#include <cstdlib> // for extern int std::atoi(const char[]);
#include <vector>
…
std::vector<double> data;
…
template<typename T1, typename T2>
void foo(T1 (*f)(const char *), std::vector<T2> &c);
…
foo(std::atoi, data);   // f is poiner to function
                        //   - taking a const char * argument
                        //   - and returning an int
                        // c is reference to std::vector<double>
                        // hence:  T1 is int and T2 is double
```

Surely it would be nice if the type deduction as far as stated in the comments of the above example could be "proven" somehow …*

*: … though this can be tricky with the standard instruments of C++. For more convenience Boost.Typeindex may be used.

Template Type Deduction Example (3)

Templated types are also deduced if they occur more than once in the argument list:*

```
template<typename T>
void bar(T (*)(const char *), std::vector<T> &c);
…
bar(std::atof, data);   // f is poiner to function
                        //   - taking a const char * argument
                        //   - and returning a double
                        // c is reference to std::vector<double>
                        // hence:  T is double
bar(std::atoi, data);   // ERROR (T cannot be both, int and double)
```

---

*: Sometimes the problem may be subtle, but can be recognized (and solved) with a systematic analysis, like in the example below:

```
// which change will make the following code compile?
template<typename T> void baz(T &val, std::vector<T> &c);
…
std::vector<double> data;
const double PI = 3.14152;
baz(PI, data);
```

More Type Deduction Scenarios

Besides the type deduction scenarios introduced here, there are some more:

Introduced with C++11 was:

- `decltype`
- perfect forwarding
- lambda captures and returns

C++14 added some more:

- `decltype(auto)`
- lambda init captures
- function return values

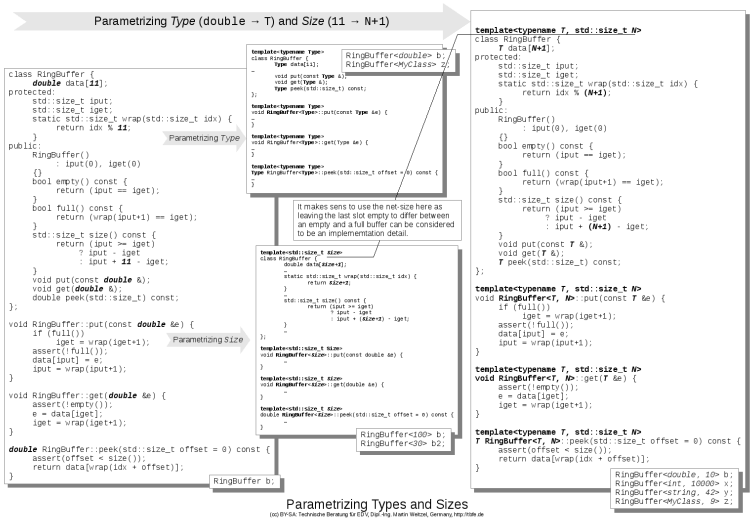
These use similar basic rule sets but with some variations* and will be covered – as far as necessary – in an adequate later chapters.

For most C++ developers it is not necessary to become the ultimate specialist for type deduction, as – especially in the simple cases – everything works as expected …

*: … thus opening the door for confusion. A good starting point to delve deeper into the topic of type deduction is the following video lecture by Scott Meyers: https://vimeo.com/97344493

Template Basics

- Parametrisierte Typen und ...
- ... Compilezeit-Konstanten ...
- ... am Beispiel einer `RingBuffer`-Klasse



Parametrisierte Typen

Ursprünglich sollte mit den C++-Templates vor allem weitgehend identischer Code reduziert werden, wenn es in verschiedenen Varianten einer Klasse oder (Member) Funktion nur um andere Datentypen geht.

- Die Funktion oder Klasse ist hierzu mit einer formalen Typ-Parameterliste in spitzen Klammern zu versehen.
- Darin werden dem Compiler **symbolische Namen** für die parametrisierten Typen angekündigt.*

Diese Namen können in der nachfolgenden Implementierung der Klasse oder Funktion überall dort verwendet werden, wo syntaktisch ein Typ stehen kann bzw. muss.

Bei der Instanziierung der Template sind in den spitzen Klammern an der entsprechenden Stelle der Parameterliste konkrete Typen einzutragen.

*: Each name is preceeded by the keyword `class` or `typename`, which may be used interchangeably with same meaning here. (But note that the two keywords have different meanings elsewhere.)

Parametrisierte Compilezeit-Konstanten

Im Rahmen einer Template-Klasse oder -Funktion können nicht nur Typen sondern auch Compilezeit-Konstanten parametrisiert werden.

Hierzu ist

- die Funktion oder Klasse mit einer formalen Wert-Parameterliste in spitzen Klammern zu versehen,
- in welcher dem Compiler **Typen und symbolische Namen** der parametrisierten Compilezeit Konstanten angekündigt werden.

Diese Namen können in der nachfolgenden Implementierung der Klasse oder Funktion überall dort verwendet werden, wo syntaktisch eine Konstante des betreffenden Typs stehen kann.

Bei der Instanziierung der Template sind in den spitzen Klammern an der entsprechenden Stelle der Parameterliste Compilezeit-Konstanten des betreffenden Typs einzutragen.*

* The usual automatic type converions take place as necessary – e.g. between arithmetic types.

Beispiel `RingBuffer`

Die Umwandlung der zunächst für einen bestimmten Datentyp und eine feste Größe implementierten `RingBuffer`-Klasse in eine Template kann größtenteils durch systematisches "Suchen und Ersetzen" erfolgen:

- Besonders günstig ist, dass der Datentyp `double` nur dort auftritt, wo in der Template-Variante der parametrisierte Typ stehen muss.
- Die Konstante `11` steht im ursprünglichen Code für die Anzahl der im `RingBuffer` maximal ablegbaren Elemente **Plus Eins**.*
  - In der Template-Variante erscheint es sinnvoll, die Nettogröße (also die maximale Anzahl der Elemente) anzugeben, die der `RingBuffer` tatsächlich aufnehmen kann.
  - Dies ist leicht realisierbar, indem bei einem Größen-Parameter `N` die Konstante `11` jeweils durch `N+1` ersetzt wird – in Ausdrücken ggf. in Klammern zur Sicherstellung des Operator-Vorrangs.

---

*: So that the *empty* and *full* state can be easily discerned without an additional flag, the buffer never gets completely filled but a single element is always left unused, if the position into which to "put" is directly behind the position from which to "get".

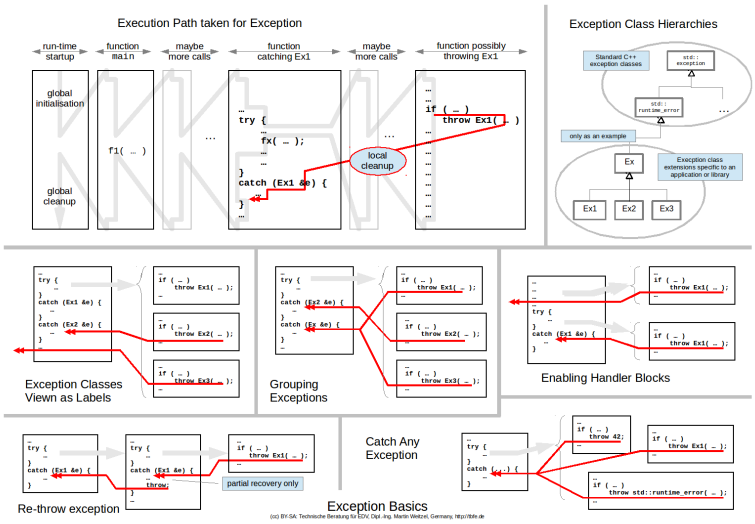
Exception Basics

- Hierarchien von Exception-Klassen

- Kontrollfluss mit und ohne Exceptions

- Exception Klassen als Label verstanden
- Gruppieren ähnlicher Exceptions
- Aktivieren der Behandlungs-Blöcke

- Unvollständig behandelte Exceptions
- Fangen aller Exceptions



Hierarchien von Exception-Klassen

Die im Rahmen von Bibliotheksfunktionen ggf. geworfenen Exceptions bilden eine Klassenhierarchie:

- An deren Spitze steht die Klasse `std::exception`.
- Davon abgeleitete Klassen sind oft als Basisklassen für eigene Exception-Klassen sinnvoll, etwa
  - `std::logic_error` oder
  - `std::runtime_error`.

Click here for Info-Graphic
ExceptionBasics
Execution Path taken for Exception

Kontrollfluss mit und ohne Exception

Solange keine Exceptions geworfen werden, folgt der Kontrollfluss den üblichen Regeln.

> Erreicht der Kontrollfluss das Ende eines `try`-Block, ohne dass eine Exception geworfen wurde, werden alle nachfolgenden `catch`-Blöcke übersprungen.

Insoweit kann das Verhalten analog zu einem Block nach `if` sehen, bei dem ebenso alle nachfolgenden `else if` und das abschließende `else` übersprungen werden, wenn die Auswertung der Bedingung `true` ergab.

Exception Klassen als Label verstanden

Sobald jedoch eine `throw`-Anweisung ausgeführt wird, stellen die den aktiven `try`-Blöcken nachfolgenden `catch`-Blöcke eine Art Label dar:

> Verzweigt wird ausgehend vom `throw` grundsätzlich im Kontrollfluss rückwärts, d.h. in Richtung auf die `main`-Funktion.

- Die Auswahl der `catch`-Blöcke erfolgt dabei
  - gemäß der dynamischen Schachtelung der Funktionsaufrufe, und
  - und innerhalb der einzelnen `catch`-Blöcke nach einem aktiven `try`-Block von oben nach unten.
- Es wird der erste `catch`-Block ausgewählt, bei dem der Typ dessen Typ zur geworfenen Exception passt.
- Gibt es keinen solchen, geht zum nächsten, gemäß der dynamischen Schachtelung folgenden `try`-Block.
- Wird dabei auch die `main`-Funktion verlassen, bricht das Programm ab.

Gruppieren ähnlicher Exceptions

Bei der Auswahl des `catch`-Blocks werden in Bezug auf die geworfene Exception prinzipiell die selben automatischen Typ-Konvertierungen berücksichtigt wie bei der Übergabe von Parametern an Funktionen.

- Insbesondere gilt das LSP, d.h. abgeleitete Klassen passen zu ihren jeweiligen Basisklassen.
- Somit lassen sich Klassenhierarchien bilden, um ähnliche Exceptions optional in einem gemeinsamen `catch`-Block fangen zu können.

Auch wenn die Klammern nach `catch` dem Konstrukt insgesamt das Aussehen einer Argumentliste geben, stellt die Verzweigung des Kontrollflusses in einen `catch`-Block **keinen Funktionsaufruf** dar.*

*: It is rather some kind of special return into the control flow of the (still) active function with the (once) active `try`-Block. But there are even more similarities to a function and a parameter specification, not only that the usual type conversions take place, but also with respect to `const` and value or reference access to the exception object thrown. Finally, two requirements imposed syntactically are that there must always be exactly one "argument" and that a `catch`-block must always be written as block, so even if it contains exactly one statement the curly braces may not be omitted.

Aktivieren der Behandlungs-Blöcke

Ein `try`-Block wird aktiv, sobald der Kontrollfluss die erste enthaltene Anweisung erreicht und deren Ausführung beginnt.

> Als Sprungziel in Betracht gezogen werden stets nur diejenigen `catch`-Blöcke, deren vorangehender `try`-Block aktiv ist.

Ein `try`-Block ist nicht mehr aktiv, wenn er explizit verlassen wird durch

- `return`*
- `break`
- `continue`

bzw. wenn die letzte enthaltene Anweisung vollständig ausgeführt wurde.

> Anschließend werden die nachfolgenden `catch`-Blöcke nicht mehr als mögliche Behandlungs-Blöcke nach einem `throw` in Betracht gezogen.

*: If a value is returned, evaluation of an expression may be part of the return. The evaluation itself takes place while technically still in the active `try`-block. But when the function has returned and its return value is only used – say in another copy c'tor outside the function that has returned – the function's `try`-block is not active any more and hence the `catch`-blocks following it are not any more possible targets.

Unvollständig behandelte Exceptions

Catch-Blöcke, welche nur eine teilweise Auflösung der Fehlersituation leisten, können die gefangene Exception erneut werfen:

```
    try {
        …
        … // code that may throw SomeException (no matter if
        … // directly, or indirectly from a function called)
        …
    }
    catch (SomeException &ex) {
        …
        … // (assuming partial recovery only)
        …
        throw;
    }
}
```

Fangen aller Exceptions

Es besteht die Möglichkeit, einen `catch`-Block für alle erdenklichen Exception "passend" zu machen:

- Hierzu sind in den runden Klammern drei Punkte (analog zu variadischen Funktionen) anzugeben.
- Ein solcher Block muss ggf. immer am Ende aller `catch`-Blöcke eines `try`-Block stehen.*

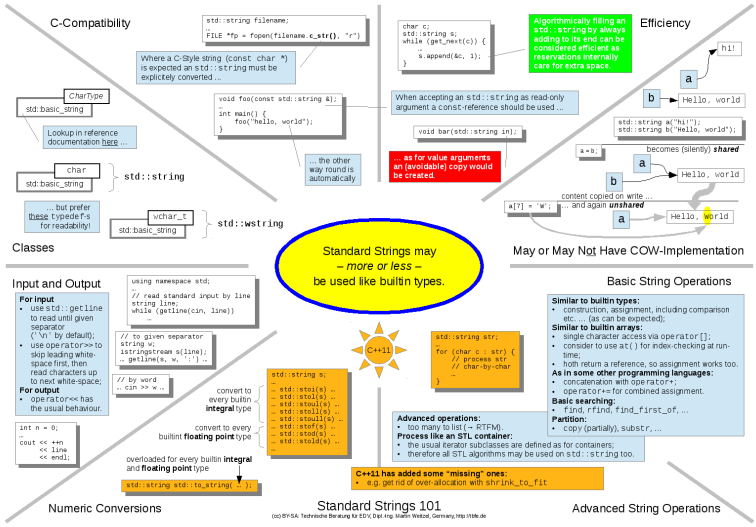
```
int main() {
    try {
        …
        …
    }
    catch (...) {
        std::cerr << "!! unhandled exception !!\n";
    }
}
```

*: This is not syntactically enforced but as "..." catches any exception and the `catch`-blocks are considered top down, it would otherwise catch anything for which a specific `catch`-block is following.

Library-Basics – Strings

- Klassen (Übersicht)
- Kompatibilität zu C
- Effizienz-Betrachtungen
- Optionales "Copy On Write"

- Grundlegende Operationen
- Weitere Operationen im Überblick
- Umwandlung von/in arithmetische Werte
- Ein- und Ausgabe



Klassen (Übersicht)

Die aus C++98 bekannten `std::string` und `std::wstring` sowie die von C++11 hinzugefügten Klassen `std::u16string` und `std::u32string` sind lediglich Typ-Definitionen ähnlich den folgenden:*

```
namespace std {
    typedef basic_string<char> string;      // since C++98
    typedef basic_string<wchar_t> wstring;  // since C++98
    typedef basic_string<char16_t> wstring; // since C++11
    typedef basic_string<char32_t> wstring; // since C++11
}
```

Weitere Details sind nachzuschlagen in:

- http://en.cppreference.com/w/string/
- http://www.cplusplus.com/string/

*: The type definitions show only half of the truth: other template arguments are a character traits class and an allocator (memory management policy). Both have been omitted in the example as they do not change the essential point to make.

Kompatibilität zu C

Die Klasse `std::string` speichert die Zeichen eines Strings in einem zusammenhängenden* Speicherbereich, der zumindest in der erforderlichen Größe, oft aber auch mit etwas Reserve auf dem Heap angelegt wird.

Ein Objekt der `std::string`-Klasse enthält **direkt** typischerweise drei Zeiger auf bzw. in ein Stück Heap-Speicher.

- Die Adresse des ersten enthaltenen Zeichens.
- Die Adresse des letzten (gültigen) Zeichens.
- Die Adresse, bis zu der Speicher alloziert ist.

Insbesondere sind die als `std::string` Inhalt möglichen Zeichen nicht (wie in C) dahingehend eingeschränkt, dass ein Zeichen mit der ganzzahligen Wertigkeit 0 (ein "NUL-Byte") die Zeichenkette beendet.

*: The C++98 standard left it to the library implementors whether to chose contiguous or non-contiguous storage though that freedom might be removed in a future version of the standard.

`std::string` als `const char *` verwenden

Wenn ein C-API `const char *` erwartet, muss ein `std::string`, der den Dateinamen enthält, entsprechend dieser Konvention übergeben werden:*

```
std::string filename;
…
… // get file name from user (or elsewhere)
…
// open file for reading, using the C-API
FILE *fp = std::fopen(filename.c_str(), "r");
```

Der obige Code ist **korrekt und risikolos**, da auf den von `c_str()` zurückgegebenen Zeiger (bzw. den darüber erreichbare String-Inhalt)

- nur innerhalb von `std::fopen` zugegriffen werden kann,
- die Variable `filename` dabei nicht verändert wird, und
- somit auch **der zugehörige Heap-Speicher derselbe bleibt**.

*: If the mode to open the file came from an `std::string` too, of course it needs to be converted similarily like that: … `std::fopen( … , openmode.c_str())` …

Risiken von `c_str()`

Bei Verwendung von `c_str()` oder dem seit C++11 gleichwertigen `data()` sollte klar sein, dass darüber Zugang zu dem für den `std::string`-Inhalt auf dem Heap angelegten Speicherplatz gegeben wird.

[!] Code wie der nachfolgende ist somit hochgradig riskant: der Zugriff auf `p` zeigt nach jeder Veränderung des Inhalts von `s` auf möglicherweise nicht mehr gültigen oder mittlerweile für andere Zwecke allozierten Speicherplatz.*

```
std::string s("see me, ");
const char *p = s.c_str();
s += std::string("feel me,");
…
s.append(" touch me, hear me");
…
```

```
const char *mammamia() {
    std::string local;
    …
    return local.data();  // as
    // of C++11 same as c_str()
}
```

*: In the example on the right hand side, dereferencing the pointer returned from the function will access deallocated heap memory (once owned by `local`) with near to 100% certainty.

`const char *` als `std::string` verwenden

Die Umwandlung eines klassischen C-Strings in ein `std::string`-Objekt erfolgt stets automatisch durch einen als Typ-Konvertierung wirksamen Konstruktor.

> **(i)** Sollen Funktionen sowohl mit (unveränderbaren) `std::string`-Argumenten als auch mit klassischen String-Literalen im C-Stil aufrufbar sein, besteht die sparsamste Lösung in der Verwendung des Argument-Typs `const std::string &`.

Eine – gemessen am zu schreibenden Code – deutlich aufwändigere Lösung ist es, Überladungen für

- `const char *`,
- `const std::string &` und
- (evtl.) `std::string &&`

bereitzustellen.*

*: On the other hand, each version could then be optimized for its argument type.

Effizienz-Betrachtungen

Typische Implementierungen der `std::string`-Klasse nutzen die folgenden Maßnahmen zur Effizienzsteigerung:

- Allozierung von Überschuss-Speicherplatz am Ende.
- Wenn nötig proportionale Vergrößerung* der Allozierung (nicht mit konstanten Zuschlag).
- Insbesondere auf 64-Bit Hardware lohnend: Small String Optimisation

*: Proportional here means that when the current allocation doesn't suffice any more it will be doubled (or made 1.5 or 1.8 times as large). This gives O(1) performance to algorithms that fill a long character string by appending single characters to the end. Increasing the allocation by a constant, fixed amount would yield much worse $O(N^2)$ performance.

Optionales "Copy On Write"

Durch diese Optimierung lässt sich insbesondere Code ohne weiteren Eingriff "nachbessern", welcher häufig `std::string`-s als Wertargument übergibt, wo eine konstante Referenz angemessen wäre.

- Das Kopieren des eigentlichen Inhalts wird dabei verzögert:*
  - Stattdessen wird ein "Merker" gesetzt, und
  - das Kopieren beim ggf. beim ersten schreibenden Zugriff nachgeholt.
- So lange nur lesend zugegriffen wird, teilen sich mehrere formale Kopien den Inhalt.

> Endet die Lebenszeit einer formalen Kopie, ohne dass es jemals zu einem schreibenden Zugriff kam, wurde das Kopieren gänzlich eingespart.

*: Copy On Write (COW) Implementierungen finden sich mittlerweile weniger häufig, insbesondere in multi-threaded Ablaufumgebungen, da ihre möglichen Performance-Vorteile (durch gelegentlich ersparte Kopien) dort oftmals geringer zu Buche schlagen als ihr Nachteil, alle Zugriffe auf den String-Inhalt durch geeignete Mechanismen (z.B. Semaphoren) aufwändig koordinieren zu müssen.

Grundlegende Operationen

Soweit diese nicht ohnehin intuitiv verständlich sind, wie

- Zuweisung mit `=`,
- Vergleich mit `==`, `!=` usw.
- Verkettung mit `+` sowie
- Elementzugriff mit `[…]`

stellen sie üblicherweise keine hohe Hürde dar.

Eine Überlegung zum Programmierstil beim Element-Zugriff könnte sein, diesen in nicht für die Performance relevantem Code konsequent mit der Member-Funktion `at()` vorzunehmen und damit undefiniertes Verhalten bei Bereichsüberschreitungen zuverlässig zu vermeiden.

Weitere Operationen im Überblick

An dieser Stelle sei auf die Ähnlichkeit zwischen `std::string` und `std::vector<char>` verwiesen, insbesondere gilt:

- Auch ein `std::string` bietet die übliche Iterator-Schnittstelle,
- womit neben speziellen `std::string` Member-Funktionen auch alle STL-Algorithmen anwendbar sind.

Was jeweils als besser verständlicher Code empfunden wird, ist mitunter auch Ansichtssache.

Das folgende Fragment liest einen `std::string s` von Standard-Eingabe und testet vor der weiteren Verarbeitung, ob ausschließlich "White-Space" enthalten ist:

```
// with std::string member function
if (std::getline(std::cin, s)
 && s.find_first_not_of(" \t") == std::string::npos)) …

// with STL algorithm (and predicate specified as lambda)
if (std::getline(std::cin, s)
 && !std::all_of(s.begin(), s.end(),
                 [](char c) { return (c == ' ' || c == '\t'); })) …
```

Umwandlung von/in arithmetische Werte

Die Umwandlung zwischen Zeichenketten und arithmetischen Werte in interner Darstellung (`int`, `unsigned`, `long`, … `double`) gehört zu den häufig zu lösenden Aufgaben.

Vielfach findet man hier noch sehr umständlichen, unzureichenden oder teils sogar gefährlichen Code:*

```
std::string tmpfilename; // fixed part, followed by sequence number
…
char *cp = const_cast<char*>(tmpfilename.c_str());
while (*cp && !std::isdigit(*cp))
    ++cp;                          // locate first digit
const int num = std::atoi(cp);  // convert digit sequence to int
std::sprintf(cp, "%d", num+1);  // and store back incremented by 1
```

*: If not obvious from reviewing the code, the fragment above has the following problems:
1. `std::atoi` converts up to the first non-numeric character only (hence a missing numeric part will not be recognized – though in some cases this might be rather a feature than a bug).
2. Some else's memory might be silently overwritten if at `cp` not enough space is available for storing `num` incremented.

`std::string` in numerischen Wert umwandeln

C++11 hat zu diesem Zweck eine Reihe neuer Funktionen eingeführt.

- Das Namensschema ist `std::stoXX` für string `to` mit einem
- Buchstaben-Code `XX` gemäß der nachfolgenden Tabelle:

Buchstaben-Code	Umwandlung nach	basierend auf
`i`	`int`	`std::strtol`
`l`	`long`	`std::strtol`
`ll`	`long long`	`std::strtoll`
`ul`	`unsigned long`	`std::stroul`
`ull`	`unsigned long long`	`std::strtoull`
`f`	`float`	`std::strtod`
`d`	`double`	`std::strtod`
`ld`	`long double`	`std::strtold`

Siehe auch:

- http://en.cppreference.com/w/cpp/string/basic_string/stol
- http://en.cppreference.com/w/cpp/string/basic_string/stoul
- http://en.cppreference.com/w/cpp/string/basic_string/stof

Numerischen Wert in `std::string` umwandeln

Für die Umwandlung von numerischen Werten in `std::string` führte C++11 die Funktion `std::to_string` mit einer Reihe von Überladungen ein.

Ihre Anwendung ist trivial und zusammen mit der auf der vorherigen Seite eingeführten Funktion `std::stoull` aus folgendem Beispiel ersichtlich:

```
std::string tmpfilename; // fixed part followed by sequence number
…
const auto n1 = tmpfilename.find_first_of("0123456789");
assert(n1 != std::string::npos);
const auto n2 = tmpfilename.find_first_not_of("0123456789", n1+1);
std::size_t nx;
const auto num = std::stou(tmpfilename.substr(n1, n2), &nx);
assert(nx == n2-n1);
tmpfilename = tmpfilename.substr(0, n1)
            + std::to_string(num+1)
            + tmpfilename.substr(n2);
```

Ein- und Ausgabe

Die Ausgabe von Zeichenketten erfolgt üblicherweise mit dem überladenen Ausgabe-Operator:*

```
std::string greet{"hello, world"};
…
std::cout << greet;
```

*: Of course, this is not a specific operator for output but an overload to the left-shift operator (as introduced in C), when the left-hand operand is an output stream. Nevertheless, especially when C is used outside the realm of embedded programming, some call `operator<<` now *output operator*.

Lesen mit `operator>>`

Der für `std::string` überladene `operator>>` liest wortweise:

```
std::string word;
while (std::cin >> word) …
```

Als Trennung zwischen den Worten gelten hier beliebig lange Leerraum-Folgen (White Space), üblicherweise (mindestens) die Zeichen:

- Zeilenvorschub (`'\n'`)
- Leerzeichen (`' '`) sowie
- horizontale und vertikale Tabulatoren (`'\t'` und `'\v'`).

Beim Lesen von `std::string`-s mit `operator>>` können in der Regel keine Leerzeilen erkannt (und speziell verarbeitet) werden, da alle Zeilenvorschübe im Rahmen des Überspringens von White Space stillschweigend verworfen werden.

Lesen mit `std::getline`

Eingaben können auch zeilenweise in einen `std::string` gelesen werden

```
std::string line;
… std::getline(std::cin, line) …
```

oder bis zu einem beliebigen Begrenzer:

```
std::string field;
… std::getline(std::cin, field, ':') …
```

Sehr flexibel ist das obige Verfahren jedoch nicht, da nur **genau ein** Begrenzerzeichen vorgegeben werden kann.

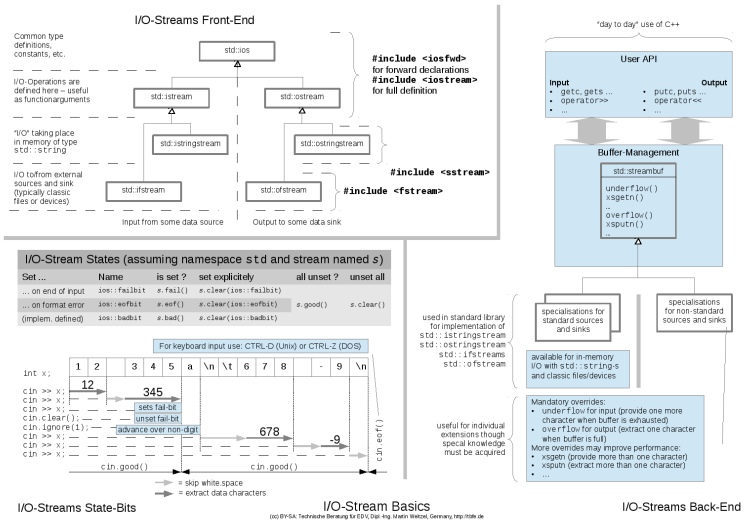
Eine Zeichenmenge – z.B. Punkt, Komma oder Semikolon – wäre oft wünschenswert, ist aber nicht möglich.*

*: While a small helper function accepting a set of delimiters shouldn't be that hard to write, this kind and much more sophisticated parsing of input patterns is possible with Regular Expressions. After having been available through Boost.Regex for a long time – regular expressions became part of the C++11 standard library (see http://en.cppreference.com/w/cpp/regex/basic_regex).

Library-Basics – IO-Streams

- Front-End und ...
- ... Back-End

- Zustands-Bits



Front-End der I/O-Streams

Das Front-End der I/O-Streams besteht aus

- der Basisklasse `std::ios`* mit einigen allgemeinen Definitionen,
- davon abgeleitet die Klassen `std::istream` und `std::ostream`, welche vor allem in Form von Referenzargumenten zur Parametrisierung von I/O-Strömen als Funktionsargumente verwendet werden,
- sowie als Klassen, von denen auch Objekte angelegt werden
  - `std::ifstream`, `std::ofstream` und `std::fstream` (File-Streams) und
  - `std::istringstream`, `std::ostringstream` und `std::stringstream` (String-Streams).

*: As with `std::string` the architecture is even more generic and the "classes" above are rather `typedef`-s for more generic template classes, which are parametrized not only in a character type but also in some other respects. This fact need not be made prominently visible if the focus is on explaining the relationship between the classes participating in the design – as it is the case here.

Gemeinsame Schnittstelle

Die von einer Applikation verwendbaren Operationen zur Ein- und Ausgabe liegen teils als Member der Klassen `std::istream` und `std::ostream` vor, teils sind es globale Funktionen.

> Zur Unterstützung benutzerdefinierter Datentypen können auch weitere globale Überladungen von `operator>>` und `operator<<` existieren.

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_ios
- http://en.cppreference.com/w/cpp/io/basic_istream
- http://en.cppreference.com/w/cpp/io/basic_ostream

Filestreams

Die Verwendung erfolgt typischerweise abhängig von der gewünschten Richtung des Datenstroms:

- `std::ifstream` zum Lesen
- `std::ofstream` zum Schreiben

Darüberhinaus gibt es auch bidirektional verwendbare Filestreams:

- `std::fstream`

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_fstream
- http://en.cppreference.com/w/cpp/io/basic_ifstream
- http://en.cppreference.com/w/cpp/io/basic_ofstream

Stringstreams

Die Verwendung erfolgt typischerweise abhängig von der gewünschten Richtung des Datenstroms:

- `std::istringstream` zum Lesen
- `std::ostringstream` zum Schreiben

Darüberhinaus gibt es auch bidirektional verwendbare String-Streams:

- `std::stringstream`

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_stringstream
- http://en.cppreference.com/w/cpp/io/basic_istringstream
- http://en.cppreference.com/w/cpp/io/basic_ostringstream

Back-End der I/O-Streams

Das Backend der I/O-Streams implementiert vor allem einen Mechanismus zur Datenpufferung.

Damit kann insbesondere

- die Übertragung von Daten in Richtung von oder zu einem permanenten Speicher in optimierten Blockgrößen erfolgen,
- welche die lesende bzw. schreibende Applikation weder kennen noch in irgend einer anderen Weise berücksichtigen muss.

Für alles weitere wird auf die Standarddokumentation verwiesen, z.B.:

- http://en.cppreference.com/w/cpp/io/basic_streambuf
- http://en.cppreference.com/w/cpp/io/basic_filebuf
- http://en.cppreference.com/w/cpp/io/basic_stringbuf

Zustands-Bits der I/O-Streams

Jeder Stream besitzt eine Reihe von Zustandsbits.

- Ist keines gesetzt ist, befindet sich der Stream im *good*-Zustand.
- Bei im Rahmen der Eingabe eines bestimmten Daten-Typs *unerwarteten (also nicht zu verarbeitenden)* Zeichen wird das `std::ios::failbit` gesetzt.
- Tritt im Rahmen der Eingabe die *End-Of-File*-Bedingung ein, wird das `std::ios::eofbit` gesetzt.
- Bei anderen – vom Standard nicht näher spezifizierten – Fehlerbedingungen kann auch das `std::ios::badbit` gesetzt werden.*

*: The usual difference between setting the *fail*- or *bad*-bit is that in the latter case there is often no (portable) way to recover, while in the former any unexpected input causing the state-switch might simply be skipped.

Verhalten der I/O-Streams abhängig vom Zustand

Solange eines der genannten Bits gesetzt ist, werden vom betreffenden Stream alle Operationen ignoriert, **ausgenommen `clear()` und `close()`**, was insbesondere bei der Verarbeitung fehlerhafter Eingaben wichtig ist:

- Beim Wechsel des Zustands bleibt die aktuelle Position im Stream – also welches Zeichen als nächstes beim Lesen einer Eingabe verarbeitet wird – wo sie beim **Auslösen** des Zustandswechsels war.
- Bei einem Fehler im Eingabeformat, z.B. wenn ein Buchstabe erscheint, wo eine Ziffer erwartet wird, ist somit das nächste Zeichen immer noch das störende, z.B. der unerwartete Buchstabe.
- Soll (mindestens) dieses Zeichen übersprungen werden, so muss
  - der Stream **zuerst** in den *good*-Zustand versetzt werden, denn
  - **erst dann** kann eine Operation wie z.B. `ignore` zum Überspringen von Zeichen ihre Wirkung entfalten.

Siehe auch:

- http://en.cppreference.com/w/cpp/io/basic_ios/clear
- http://en.cppreference.com/w/cpp/io/basic_istream/ignore

Exceptions bei Zustandswechsel

Wahlweise kann das Verhalten eines Streams so eingestellt werden, dass

- bei jedem Zustandswechsel und
- allen (versuchten) Operationen außerhalb des *good*-Zustands

eine Exception geworfen wird:

```
// Excerpt from a hypothetical "forever running" TCP-Client
std::ifstream from_server;
… // somehow establish connection through TCP/IP-Socket
from_server.exceptions(std::ios::badbit
                     | std::ios::eofbit
                     | std::ios::failbit);
try {
    for (;;) {
        std::string command_string;
        std::getline(from_server, command_string);
        … // process command_string
    } /*notreached*/
}
catch (std::ios_base::failure &e) {
    … // socket connection closed and/or data transfer failed
}
```