C++11 Extensions (and some related Boost)

- General and `const`-qualified Pointers
- Classic References (Lvalue References)
- Comparison of Pointers and Reference
- Rvalue References (introduces with C++11)
- C++11 Brace Initialisation
- Template Basics
- Perfect Forwarding
- Range (-based) `for`-Loops

General and `const`-qualified Pointers

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

For a pointer `const` my refer to **the pointer itself** …

```
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!)
```

… or the memory location **reachable via the pointer**:*

```
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 = …;`).

Classic References (Lvalue References)

The classic example is a function swapping the contents of two variables:

```
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);
…
```

On the left pointers are used and addresses are handed over explicitly, on the right arguments are defined as references, causing implicit dereferencing inside `swap` and handing over addresses at the call site..[]

*: So the difference is mostly notational (simplified source code with references), while generated machine instructions are typically the same, though there is no strict rule enforcing this. E.g. depending on compile time debug options different safety checks could be generated for pointers and references.

Initialisierung von Referenzen

By concept references always denote existing memory locations,* therefore they need to be initialised.

> After its definition and initialisation a reference cannot be modified so that it subsequently refers to a different memory location.

From that point of view references are like constant pointers, which may **not** be modified themselves, though referenced memory can be read **and** written.

```
int v1, v2;  // some variables ...
int &r = v1; // reference is initialized
```

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;`

Constant References

A `const` qualifier for a reference always effects memory access via the reference.

> References which are `const`-qualified only allow reading memory, even if the referenced memory (or variable) is writable.

In the following example the variable `v` may be modified directly and via `r1`, but is read-only when accessed via `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)
```

*: 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).

Comparing Pointers to References

References may be viewed in two ways:

- **Either** an alternative syntax for pointers, which
  - during initialisation expects an lvalue of which the address is taken (i.e. "&" is automatically and invisibly applied);
  - on each **use** implies the dereferencing operation on each use (i.e. "*" is automatically and invisibly applied).
- **Or** an alias name for an existing memory location with a matching type.

> Taking inot account the invisibly applied operations to references, then there are usually the same machine instructions generated for references as are for pointers – the difference is in the syntax for initialisation and access.*

*: Hint: When compiling with `g++` or `clang++` you may want to use the command line option "`-S`" (upper case) and have look at the assembler code stored in a file with suffix "`.s`" (lower case).

Rvalue Referenes

C++11 introduced Rvalue References. They may only be initialised by expressions, i.e. memory locations introduced to hold temporaries with no other access path.

Following are the most important rules:

```
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)
```

The main use for rvalue references are to overload functions differently for arguments denoting **either** existing **or** temporary objects, and again the main use for this is to implement Copy Constructor and Assignment differently from Move Constructor and Assignment.

*: As a function argument an rvalue reference is in most any case **not** `const`-qualified, as the temporary reachable over it needs to be modified (within the limits that the destructor needs still to work correctly). In contrary a classical reference typically **will** be `const` qualified if introduced for improved performance of read-only accesses.

C++11: Move Semantics

Move semantics provide the solution to two problems that could not (always) be avoided in C++98:

- Efficient use of *value types as function return values*, e.g. if they represent large containers.
- Implementing types that are *movable* but not *copyable*.*

> **(i)** Even before C++11 in many practical cases the leeway given to a compiler to apply RVO and NRVO could achieve much to **return large data structures by value efficiently**.

But as there is no guarantee in this respect, the usual recommendation for C++98 was to hand-out large containers via reference arguments, not by return value – and this recommendation should possibly be followed even in C++11.

---

*: With C++98 there is no real solution to the problem to differentiate between moveable and copyable types. Even if `operator=` stays undefined and overloaded global functions `assign` and `move` were used consequently, especially in type-generic code of template implementations something in the vein of Perfect Forwarding could not be achieved, at least not with as little code as can now.

Overloading for Rvalue References

Move semantics heavily build on Rvalue References defined with a double ampersand:

```
void foo(const T &classic_reference) { … } //first
void foo(T &&rvalue_reference) { … } //second
```

With the above two overloads C++11 would bind

- the first `foo` to arguments that are plain variables (including `const` qualified variables),
- the second `foo` to arguments that are temporaries which will be destroyed soon after use.*

```
T a; const T b; extern T bar();
foo(a);     // calls first
foo(b);     // calls first
foo(bar()); // calls second
foo(a+a);   // calls second -- provided T supports operator+
```

*: Such as function calls, but also including constants and expressions (with some reasonable exemptions).

Copyable and Movable Types

Instances of the following class will be both, copyable and movable:

```
class MyClass {
    …
public:
    MyClass(const MyClass &); // classic copy constructor and ...
    const MyClass& operator=(const MyClass &); // copy assignment
    …
    MyClass(MyClass &&);        // C++11 move constructor and ...
    const MyClass& operator=(MyClass &&);      // move assignment
}
```

By supplying both of move and copy support, or only the one or the other, or none at all, instances of `MyClass` can easily be made

- **Copyable** and **Movable**,
- **Copyable** but not Moveable,
- not Copyable but **Movable**, or
- neither Copyable nor Moveable.

C++11: `default`-ed and `delete`-d Operations

C++11 furthermore provides a particular syntax to request or forbid compiler generated constructors and assignments, making it easy to write a class that supports the required behavior:

```
class MyClass {
    …
public:
    MyClass(const MyClass &)                    = delete;
    const MyClass& operator=(const MyClass &)   = delete;
    …
    MyClass(      MyClass &&)                   = default;
    const MyClass& operator=(      MyClass &&) = default;
 };
```

It is probably easy to spot that instances of the above class will be moveable but not copyable and what needs to be changed if that should be different.

In case the default implementation provided for the above operations is not appropriate, then of course a specific implementation can be supplied.

Boost: Noncopyable

As C++ always generates a copy constructor when none is specified,* the usual technique is to *declare-but-not-implement* the unwanted operation.

Via deriving from `boost::noncopyable` the intent can be made more obvious (and code a bit more compact):

```
class MyClass : boost::noncopyable {
    …
    … // whatever (but no need any more to define
    … // operations that never get implemented)
    …
};
```

*: Note that with C++11 the rules changed in so far as **no default copy-constructor will be provided if a move-constructor is provided**, and the same holds for copy- and move-assignment. The reasoning behind that rule is that as soon as a specific behavior is necessary for one, copy or move, it will probably also be the case for the other.

C++11: Uniform Initialization

C++ traditionally had many forms of initialization, some of which were limited to certain contexts:

```
int x = 0;                      // traditional style
const std::string greet("hi"); // constructor style
struct s {
    int a;
    char z;
} v = { 42, '!' };              // aggregate initialization
std::string empty();            // INVALID (as initialization)
unsigned u = unsigned();        // not common but valid (in C++)
```

Since C++11 curly braces may be used in any initialization context:*

```
int x{0};                       // explicit zero initialization
const std::string greet{"hi"}; // initialization by constructor
string empty{};                 // valid for default constructor
unsigned u{};                   // implicit zero initialization
```

*: Compared to classic initialization some rules have slightly changed: E.g. if the value of the initializing expression cannot be represented in the initialized variable, this is a compile time error.

C++11: Initializer Lists

Initializer lists are sequences of comma-separated values enclosed in curly braces.

- They are valid wherever a function accepts an argument of type `std::initializer_list`.
- This includes many constructors for standard containers:*

> **[!]** A few usage forms introduced ambiguities for which C++11 defined disambiguating rules – sometimes little intuitive ones.

```
vector<short> primes({ 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 });
const map<string, string> words = {
    { "zero", "null" }, { "one", "eins" }, { "two", "zwei" },
    …
};
vector<int> x{3};  // A vector initialized with a "list" just
                   // holding a single 3?
                   // Or rather a vector sized to 3 elements
                   // that shall be default-initialized?
```

Boost: Value Initialized

Using the correct initialization syntax can pose a problem* at times:

```
template<typename T> void foo() {
    T local = … // ???
    …
}
```

- A plain `T local;` would default initialize classes but leaves basic types uninitialized.
- Classic style `T local = 0;` would only zero-initialize basic types.*

Boost.Value_initialized provides a utility template, causing either zero or default initialization, depending on the type of `T`:

```
value_initialized<T> local; // uniform use of the Boost solution
T local = T();              // a (somewhat lesser known) C++98 solution
T local{};                  // C++11 uniform initialization as solution
```

*: Actually the state of affairs is a bit more complicated: The above would also work if `T` were a class with a (non-`explicit`) constructor taking an argument type to which `0` can be converted. Besides all arithmetic types this include `bool` (`0` converted to `false`) and any pointer type (`0` converted to `nullptr`).

Boost: Container Initialization

Boost.Assign provides some operator overloading to allow a more readable initialization syntax for sequential and associative containers.

Overloaded `operator,` and `operator+=` help with sequential containers:

```
vector<int> primes;
primes += 2, 3, 5, 7, 11, 13, 17, 19, 23, 29;
```

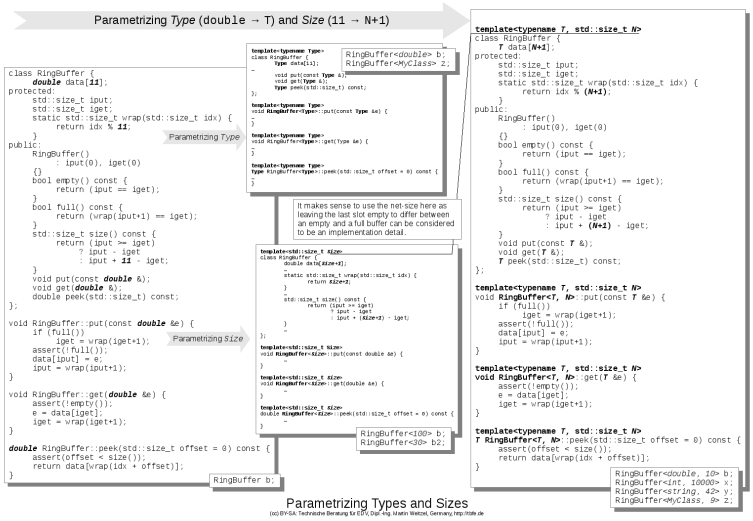
For associative containers there is a somewhat tricky overload of `operator()` (function call):

```
map<string, string> words;
words(("one")("eins"))
     (("two")("zwei"))
     …
     (("nine")("neun"))
     ;
```

Compared to C++11 initializer lists the above not only looks clumsy but also has the draw-back that no `const`-qualifiers are possible.

Template Basics

- Parametrized Types and ...
- ... Compile-Time Constants ...
- ... Demonstrated with a `RingBuffer` Example



Parametrized Types

C++-Templates were designed with the original intent to reduce mostly identical **source code** when different variants of a class or (member) function only differ in data types.

- The function or class will take a formal type parameter list in angle brackets,
- introducing **symbolic names** to the compiler that represent types which will later be specified concretely.

> The symbolic type names may be used anywhere in declarations and in the implementation of the class, where a type is syntactically permitted.

When instantiating a **class template** concrete types need to be named at the corresponding position in the angle brackets while for a **function template** such types are often visible (and deduced) from the types of arguments given by the caller.

---

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

Parametrized Compile-Time Constants

Besides types a template class or functions may also parametrize compile-time constants.

For that purpose

- the function or class will take a formal value parameter list in angle brackets,
- introducing **types and symbolic names** to the compiler that represent constants of their given type.

> The symbolic names may be used anywhere in the class implementation (usually following) where syntactically a constant of the given type is permitted.

When instantiating a (class or function) template concrete constants need to be named at the corresponding positions in the angle brackets.*

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

Example `RingBuffer`

Turning `RingBuffer`-class originally implemented with given (fixed) type and size into a template is straight forward as it boils down to some rather systematic "find and replace":

- This is especially true as `double` occurs in the source only where the (now) parametrized type symbol `T` needs to appear.
- The constant `11` specifies (in the original code) the maximum number of elements in the `RingBuffer` **plus one!***
  - From the perspective of the `RingBuffer`-s user it makes more sense to supply the net size `N` (number of elements that can actually be held).
  - This can be easily achieved by replacing each occurence of `11` (in the original code) with `(N+1)` (in the template).

*: 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".

Perfect Forwarding

A special use case for templates is *Perfect Forwarding*, which basically is about keeping the "Rvalue-ness" of a function argument until it is used.

It can be applied

- either in cookbook style or
- with first explaining (and consequently trying to grasp) a lot of its background.

> One key point with both approaches is to understand that the rvalue reference declarator (&&) is special if used with a template argument.

Scott Meyers has even coined a special term for it: Universal References

Perfect Forwarding - Cookbook Style

Perfect forwarding is achieved by applying the following recipe:*

```
template<typename T1, typename T2>
void foo(T1 &&arg1, T2 &&arg2) {
    bar(std::forward<T1>(arg1));
    baz(std::forward<T2>(arg2));
};
```

If there are overloads of `bar` and `baz` optimized for lvalue and rvalue arguments, the appropriate one will be called.

---

*: Perfect forwarding always requires the use of templates, even if it is applied to cope with the exponential explosion of overloads necessary if only arguments of known types are to be forwarded, like here:

```
// Assume objects of these classes can be arguments to foo, which it
class MyClass { … }; // it forwards to bar and baz and the caller may
class AnOther { … }; // either supply lvalues or rvalues as argument:
…
void foo(const MyClass  &arg1, const Another  &arg2) { … }
void foo(const MyClass  &arg1,       Another &&arg2) { … }
void foo(      MyClass &&arg1, const Another  &arg2) { … }
void foo(      MyClass &&arg1,       Another &&arg2) { … }
```

Perfect Forwarding - The Background

The background of perfect forwarding combines three essential pieces:

1. For a template argument `T&&` there is a special deduction of `T`:
   - `T` is *reference* for lvalue arguments,
   - otherwise it is the plain argument type (as usual with `&` and `const` stripped away).
2. Reference collapsing defines what happens if two reference declarators are applied in a row:
   - & and & → &
   - & and && → &
   - && and & → &
   - && and && → &&
3. `std::forward<T>(expr)` statically casts `expr` to `T&&`.

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


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

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

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).

Callables, `function`, and Lambdas

This section covers *Callables*, which in classic C++ (until C++11) subsumes

- classic C-style function pointers and
- C++-style Functors.*

As all of these are different types C++11 recognized the need for Type Erasure and introduced

- `std::function`,
- originally developed as Boost.Function.

Furthermore C++11 Lambdas now provide a shorthand to define functions at the point of use.

There has also been a "pure-library solution" provided by Boost.Lambda which may be a bit briefer in simple cases but will most probably be superseded now by the C++11 solution.

---

*: Functors are classes that overload `operator()` what – to plain functions – mainly adds the option for further parametrisation (in addition to the caller-supplied arguments).

Using `std::function` vs. Fully Generic Types (1)

A predicate in a (templated) algorithm could be restricted to a callable type via `std::function`:

```
template<typename T1, typename T2>
T2 filter(T1 beg, T1 end, T2 to,
          std::function<bool(typename T1::value_type)> pred) {
    while (beg != end) {
        const auto &val = *beg;
        if (pred(val))
            *to++ = val;
        ++beg;
    }
    return to;
}
```

If `T1` and `T2` denote types (maybe different to each other)

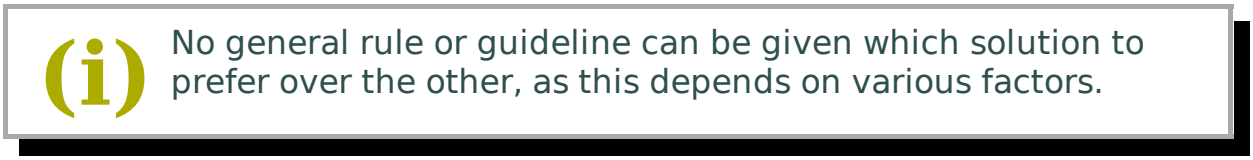
- which do not vary for many calls
- that all use a different predicate

there will be **only one single** instantiation of this function. (I.e. the instantiation will not depend on the actual predicate.)

Using `std::function` vs. Fully Generic Types (2)

In contrast to the previous example, with the following template the compiler is able* to inline the predicate evaluation, resulting in some "code bloat":

```
template<typename T1, typename T2, typename T3>
T2 filter(T1 beg, T1 end, T2 to, T3 pred) {
    while (beg != end) {
        const auto &val = *beg;
        if (pred(val))
            *to++ = val;
        ++beg;
    }
    return end;
}
```



*: Whether the actual predicate evaluation will be inlined also depends on other factors, like its definition is visible (and maybe explicitly made `inline`), or whether it is a classic function for which at the call site only an `extern` declaration is known. Also note that different compilers may vary in handling the above and the final result may depend on the optimization level requested.

Boost: Callables

C++11 `std::function` emerged from Boost.Function.

There are few differences, the most important is that `boost::function` tried to be compatible with older compilers that did not fully support the required *Preferred Syntax*.

So Boost supplied its *Portable Syntax* as an alternative:

```
boost::function0<void> f1;     // no args, returning nothing
boost::function1<int, int> f2;     // int arg, returning int
boost::function1<double, const char *> f3;
boost::function3<double, const char *, const char**, int> f4;
```

- It encodes the return type as first template argument,
- followed by the argument types (if any), and
- reflects the **number of arguments** in the class name.

C++11: Lambdas

Introducing Lambdas with C++11 was a major step to bring C++ at level with many other modern programming languages, in which

- functions not only were made "*First Class Citizens*" but
- it is also possible to specify a function body **at its point of use**,* especially as argument to some other function call.

The general definition syntax

- starts with a capture list in square brackets,
- followed by an argument list in round parenthesis,
- followed by the function body in curly braces.

*: This is why lambdas are also known as *Function Literals*.

Lambda 101 – Definition Syntax Example

Building on the `filter` algorithm from a prior slide the predicate could be supplied directly and clearly visible at the call site:

```
std::vector<double> data, result;
… // fill data
filter(data.begin(), data.end(), std::back_inserter(result),
       [](double e) { return (e < std::sqrt(2.0)); }
};
```

Note that the above will work with either version of filter, the one with the predicate parametrized to any type and the one with the predicate limited to an appropriate callable.

*: An algorithm like this is available as `std::copy_if` in C++11. (It was actually missing from C++98 where `std::remove_copy_if` had to be used with the predicate logic inverted.)

Lambda 101 – Capture Lists (Motivation)

It should be understood that for any function (e.g. `filter`) expecting some other function as argument,

- the function handed over by the caller (i.e. as argument to `filter`)
- must be callable by the coded inside (i.e. the implementation of `filter`).

Therefore it is not possible to hand over additional arguments directly:

```
filter(data.begin(), data.end(), std::back_inserter(result),
       [](double e, double max) { return (e < max); }
};
```

[!] This will not compile because the lambda not matches the expectation `filter` has about the use of its fourth argument.*

*: How the error manifests in a compiler diagnostic is a different issue: if – after some type deduction – `filter` had an argument of type `std::function<bool(double)>` the compiler will complain at the call-site not being able to initialize the predicate argument from what is specified; for a fully generic template it will rather complain inside the code of the template where the actual call is coded.

Lambda 101 – Capture Lists (Example)

In a capture list variables from the local context may be named.

Then in the code generated for the lambda that argument is transfered via a special path:*

```
double max;
std::cin >> max;
…
filter(data.begin(), data.end(), std::back_inserter(result),
       [max](double e) { return (e < max); }
};
```

So far this presentation only tried to give some first clues about the purpose and basic use of lambda capture lists.

> There are many more details which have not been covered yet, like handing over references in the capture list or some shortcuts for it.

Please lookup more information in the relevant reference documentation.

---

*: If you are curious about that you will get an idea when Classic C++ Function Objects are covered.

Lambda 101 – Beware of the Pitfalls

For efficiency reasons C++11 does **not demand any special rules for stack unwinding** when a lambda captures a local context by reference:*

```
function<void()> foo(int n) {
    return [&n]() { std::cout << n; }
}
…
foo(42)();
```

[!] Therefore the above code fragment steps into the area of undefined behavior.

- Of course, the situation might go unnoticed for a long time as the correct value just happens to be in the expected memory location …
- **… until, some day, a completely unrelated change is made!**

*: It is basically a similar situation as returning the address of a local variable from a function, which is undefined behavior since the first days of C. But while most any decent compiler will warn about this, the problem shown here most often goes unnoticed. It can be expected that code like above will trigger a warning too when future C++ compilers improve their checks in this respect.

Classic C++ Function Objects

Prior to C++11 lambdas – in C++98 and C++03 – code that can now be written elegantly with lambdas had to be written with Functors:*

```
struct LessCompare {
    const double limit;
    LessCompare(double lim) : limit(lim) {}
    bool operator()(double e) { return (e < limit); }
};
…
double max;
std::cin >> max; // only as example (max not known at compile-time)
…
filter(data.begin(), data.end(), std::back_inserter(result),
       LessCompare(max)
);
```

*: To avoid unnecessary complexity the `LessCompare` functor was not written as a template here. Of course this could have been easily done (and in practice probably would have done) to make the functor applicable to any type supporting `operator<()`.

Boost: Lambdas

As lambdas were missing for a long time from the C++ language proper, it was tried to emulate them via the library.

To rewrite the example that has been used a number of times:

```
#include <boost/lambda/lambda.hpp>
…
filter(data.begin(), data.end(), std::back_inserter(result),
       boost::lambda::_1 < max);
```

- The innocuous looking `boost::lambda::_1` above, together with a clever overload of `operator<`, triggers the complex template based machinery.*
- Last and finally a functor is created that overloads `operator()` to be callable with a single argument, returning `true` if the argument is less than `max`.

*: If you are curious to learn the details try to get familiar with Expression Templates first, because these build the underlying general mechanism.

Boost: Lambda Details

While C++11 lambdas now provide a much more general and flexible solution, it can be argued that Boost lambdas are less blatant and in many cases of practical relevance can be create with much fewer key strokes.*

> This is especially true if the namespace `boost::lambda` is opened via `using` directives, because then the occurrence of the plain identifiers `_1`, `_2`, or `_3` in an expression are sufficient to trigger the mechanism.

```
std::vector<double> data;
…
sort(data.begin(), data.end(), (_2 < _1)); // sort data in reverse
```

The above example, rewritten for C++11 lambdas, illustrates the point:

```
sort(data.begin(), data.end(), [](double lhs, double rhs) {
                                   return (rhs < lhs);
                               }); // sort data in reverse
```

*: Whether or not this is considered to be an issue to guide a decision pro or contra Boost lambdas may also depend on the capabilities of an IDE, which – if properly configured – might insert source code templates for frequently used C++11 lambdas with a keyboard shortcuts.

Boost: Lambda Beyond Trivial Use

Beyond the trivial use in pure expressions – and even there sometimes – using Boost Lambdas (or rather the expression templates behind them) can quickly get intricate.

While it is well possible to translate the following into a Boost Lambda, you probably will not want to do it when you can avoid it:*

```
vector<double> data;
…
int line{0};
std::for_each(data.begin(), data.end(),
              [&line](double e) {
                  std::cout << ++line << ':' << e << '\n';
                  if (line % 10 != 0) std::cout << "---\n";
              });
```

*: Of course, if you really have to back-port the above code written for C++11 to some older compiler it may be good to know that Boost Lambdas stretch far beyond the simple, expression-like use cases and all the typical flow control directives are supported.

C++11 vs. Boost Lambdas

Which one to prefer cannot be generally decided:

- For very simple expressions Boost Lambdas may be considered as an alternative.
- On the other hand, as soon as the situation is a bit more involved, correct use of Boost Lambdas quickly becomes tricky (at least).

Generally speaking with Boost Lambdas there is a steep learning curve beyond the trivial cases, especially when more flow control is necessary as a single expressions can provide.

> It will not make much sense to rewrite all usages of Boost Lambdas now that C++11 lambdas are available, but rewriting the complicated ones to better comprehensible C++11 lambdas may be worth a consideration.*

Exceptions

- Exception Basics
- Exception Guidelines
- C++11 `noexcept`
- Boost.Exception

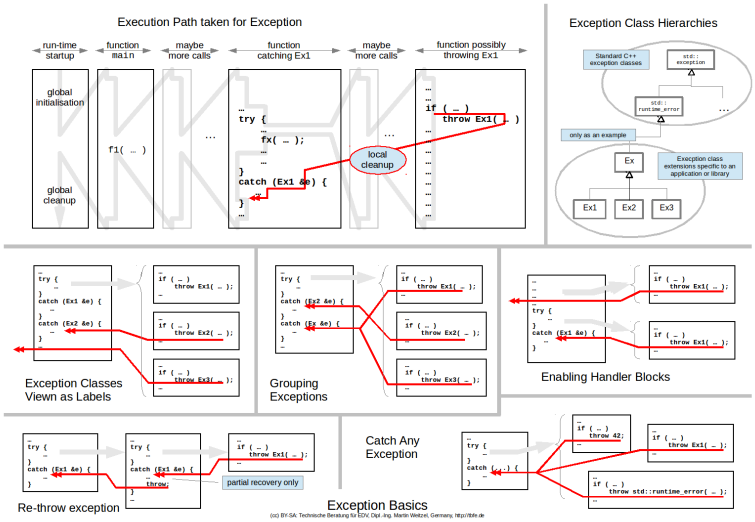
Exception Basics

- Hierarchical Exception Classes

- Flow of Control with and without Exceptions

- Understandig Exception Classes as Labels
- Grouping Related Exceptions
- Activating Re-Entry Points

- Re-Throwing Exceptions
- Catching Any Exception



Exception Class Hierarchies

Exceptions thrown by library functions build a class hierarchy:

- The common base is the class `std::exception`.
- Derived from this are the following two which may also be a good choice to extend the hierarchy for specific purposes:
  - `std::logic_error`
  - `std::runtime_error`

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

Flow of Control with and without Exceptions

When no exceptions are thrown, flow of control is as usual:

> If the end of a `try`-block is reached, all subsequent `catch`-blocks are skipped over.

Insofar behaviour is analogous to an `if`-block which skips all subsequent `else if` and the final `else`, when the condition (of the first `if`) holds true.

Understanding Exception Classes as Labels

When a `throw`-statement is executed, all the `catch`-blocks after an active `try`-block are like labels, i.e.:

> Control flow always **branches back** (removing stack frames) into the direction of the main function.

- A `catch`-block is chosen
  - according to the dynamic nesting of function calls, **and**
  - **top - down** among all `catch`-blocks subsequent to an active `try`-block.
- The selected `catch`-Block is the first one with an exception type compatible to the exception thrown.
- If there is no such `catch`-block, the next active `try`-blocks (closer to the main function) is considered.
- If none is found down to the main function program execution stops.

Grouping Related Exceptions

Compatibility of the exception thrown and the exception named in a `catch`-block is – with respect to automatic conversions that may be applied – decided in the same way as for arguments on function calls:

- Especially the LSP is effective, i.e. publicly derived classes are compatible with (any of) their base classes.
- Therefore class hierarchies make sense for exceptions too, as related exceptions then may (optionally) be caught in a common `catch`-block.

> Even in the parentheses following `catch` make the construct look like a function argument list, having the control flow branch back to a `catch`-block is different from a function calls

Rather `catch`-blocks are points for re-entry into a still active function.*

*: 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.

Activating Re-Entry Points

A `try`-Block becomes active as soon as the control flow reaches the first contained statement.

> Targets for branching-back in case of an exception throw are only `catch`-blocks following active `try`-blocks.

A `try`-Block is not any longer active after it is left by

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

or when last contained statement has completed execution.

> The `catch`-blocks following this `try`-block are not any longer considered as targets for thrown exceptions.

*: 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.

Re-Throwing Exceptions

It is not unusual that a `catch`-blocks may only partially resolve the problem indicated by an exception thrown. Then the exception must be re-thrown from that `catch`-block.

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

Catching Any Exception

It is possible to specify a `catch`-block to match any exception:

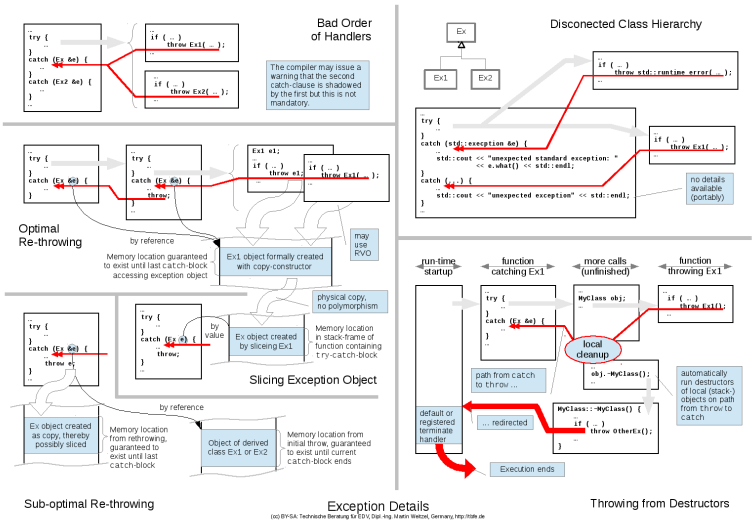
- In the parentheses (analogous to variadic functions) three dots need to be specified.
- If present, such a block must be the last of all `catch`-blocks following some `try`-block.*

```
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 as label-like targets top down, it will otherwise catch thing for which a specific `catch`-block follows.

Guidelines for Using Exceptions

- Disconnected Class Hierarchy
- Wrong order of `catch`-Blocks
- Problematic Re-Throwing
- No Exceptions from Destructors



Disconnected Class Hierarchy

For exceptions classes from a disconnected class hierarchy there is **no** possibility for catching these exceptions in generic code:

```
int main() {
    using namespace std;
    try {
        …
        … // ordinary application
        …
        return EXIT_SUCCESS; // macro in <cstdlib>
    }
    catch (exception &e) {
        cerr << "terminated by standard exception: "
             << e.what() << endl;
    }
    catch (...) {
        cerr << "terminated by unknown exception" << endl;
    }
    return EXIT_FAILURE; // macro in <cstdlib>
}
```

(Falsche) Behandlungsblock-Abfolge

With a wrong order of `catch`-blocks

- an earlier, **more general** block
- **hides a subsequent** (more specific) block.

Problematic Re-Throwing

Rethrowing an exception is sub-optimal if the original thrown object needs to be copied.

Furthermore, in case of a derived exception object, this will cause slicing.

In most any case slicing is not what makes sense.

Hence copying should be avoided:

- Exceptions in a `catch`-block should be specified as reference.
- If a `catch`-block recovers from an exception only partial, it should throw the received object with `throw;`.

Optimal Re-Throwing

If the scheme shown below is applied, the originally received exeption object will be re-thrown without making any copy (and hence also no slicing):

```
try {
    …
    … // code that may cause an exception
    …
}
catch (SomeException &ex) {
    …
    // only partial recovery
    throw;
}
```

A `throw`-Statement without argument following may not only be used in a `catch`-block but also in a function that is directly or indirectly called from such a block.*

*: That way common code of several `catch`-blocks may be placed in a helper function even if this function has a non-trivial flow of control and the exception caught is not only re-thrown at the end.

Compiler Optimisations

When entering a `catch`-block

```
catch (SomeException ex) …
```

and when leaving a `catch`-block

```
    throw ex;
```

would technically require to copy the exception object.

- Within the limits of the semantics specified by the C++ standard there are possible optimisations.
- This implies slicing and not executing private copy constructors …
- … but a copy constructor **need not** be actually executed.

When an exception is thrown, using RVO is a typical optimisation.*

*: Also NRVO might be considered but in practical cases this will be rare applicable as typically exception objects are constructed in the `throw`-statement with a constructor call.

No Exceptions from Destructors

[!] Exceptions thrown from destructors **will cause program termination** if the destructor is run during stack-unwinding as part of an ongoing exception handling.

Therefore, if a destructor needs to use operations that might throw, its body should be wrapped in a `try`-block, catching all exceptions:*

```
class MyClass {
    …
    ~MyClass() {
        try {
            … // whatever needs be done
        }
        catch (...) {
            if (!std::uncaught_exception()) throw;
        }
    }
};
```

*: The example shows how the caught exception might be re-thrown if the destructor is **not** run as part of some exception handling.

C++98 `throw`-specifications Deprecated

Most guidelines from C++ experts recommend **not to use** `throw`-specifications as introduced with C++98.

> C++11 turned `throw`-specifications into a *deprecated language feature*.*

*: This means they might be completely removed from the language, though experience tells that compilers will still support deprecated features, maybe requiring special command line options for backward compatibility.

C++11 `noexcept`-specification

The effect is as follows:

- If some function declared `noexcept` will directly or indirectly throw an exception the **program will be terminated** – may be after some user defined handler code.
- Hence in **callers** of `noexcept` functions the compiler can omit otherwise necessary book-keeping for stack-unwinding.*

Currently there is little experience with respect to `noexcept` and how they relate to code generation and overall program quality, so there are not (yet) many guidelines when and when not to use `noexcept`.

*: Be sure to understand this: In the callers context exceptions from "`noexcept`" functions may indeed stay unconsidered ... but **not** as the possibility of an exception is completely excluded but **because the callee will never return** to the caller's context!

Conditional Freedom from Exceptions

Besides what was explained on the previous page `noexcept` is also a compile-time function which especially in templates is useful to **conditionally** freedom from exceptions.

- That way some function may express that exceptions will only occur if some (unknown) called function throws.
- This allows at compile-time to decide if some piece of code bears the risk to throw.
- Using techniques from meta-programming this allows library functions to chose a different (probably non-throwing, then) implementation to make itself safely made `noexcept`.

Examples for `noexcept`

Some easy calculation – exceptions are completely excluded:

```
float fahrenheit_to_centigrade(float temperature) noexcept {
    return 9.0*temperature/5.0 + 32.0;
}
```

Some vital functionality – no chance to revover, better crash completely:

```
extern void may_fail_catastrophically(int = 42) noexcept;
```

An exception from `bar` is only thrown it `T::foo` throws:*

```
template<class T>
void bar(const T &arg) noexcept(noexcept(arg.foo())) {
    … // an exception can neither occur here ...
    arg.foo();
    … // ... nor can an exception occur here
}
```

*: The C++11-syntax is that ugly and requires nested `noexcept`-s.

Boost Exception

Boost.Exception documents the purpose of that library in the section Motivation as follows:

> Traditionally, when using exceptions to report failures, the throw site:
>
> - creates an exception object of the appropriate type, and
> - stuffs it with data relevant to the detected error.
>
> A higher context in the program contains a catch statement which:
>
> - selects failures based on exception types, and
> - inspects exception objects for data required to deal with the problem.
>
> The main issue with this "traditional" approach is that often, the data available at the point of the throw is insufficient for the catch site to handle the failure.