C++ Refresh

An Refresher's Overview on C++ Based on ISO/ANSI-Standards C++98, C++03 und C++11

Durchführung:
Dipl.-Ing. Martin Weitzel
Technische Beratung für EDV
http://tbfe.de

Im Auftrag von:
MicroConsult
Training & Consulting GmbH
http://www.microconsult.com

Agenda

This agenda and the following pages show a proposed path through the topics selected for the next two days.

- The C/C++ Preprocessor
- C++-Basics – General Extensions to C
- Object-Oriented Programming
- Introduction to Exceptions
- Dynamic Memory Allocation
- Introduction to Templates
- Library Overview (Strings, IOstreams, STL ...)
- Optional: C++11 Generalized Callables
- Optional: C++11 Concurrency Basics
- Optional: Library Extensions since C++11
- Appendices and Info-Graphics

Variations can be in the beginning, e.g. which of the optional parts are of most interest and should be given priority – if time allows their coverage.

Interrupts Enabled

Some parts of this training are meant to be given really

"*Hands On*"

i.e. your hands on the keyboard, while following the explanations ... feel free to ask questions anytime and push coverage of C++ features – including the live examples – into the direction that most satisfy your needs.

> Nothing is absolutely "fixed" here – this course is given by a human, not a book that keeps silent if you do not understand something.

Online vs. Offline Resources

You may already have received this presentation printed on paper. This is because most people find it still easier to make annotations on paper.

If you only have an electronic version you may print it out yourself.

Note that any version of this document – printed or electronic and *including* all copies you make yourself – are distributed under a

Creative Commons BY-SA License

Basically this means you need to

- keep the original author's name and organization in the page footer
- license any derived work you publish in the same way.

(i) For more details please see here:
https://creativecommons.org/licenses/by-sa/3.0/
Weitere Details finden Sie hier:
https://creativecommons.org/licenses/by-sa/3.0/de/

Online Version of this Presentation

You may download an online version of the presentation from the internet:

TBD

If you have unpacked the files be sure the following is included:*

- `Presentation/pages.html`
- `Presentation/remark.min.js`
- `Presentation/CSS/styling.css`

This is the minimum necessary to view the presentations by opening `pages.html` in any recent browser, with JavaScript enabled.

*: The whole presentation consists of more files, most importantly in the sub-directories `Infographics` and `PNG`. The former are large graphics on used for extended (but mostly optional) explanations. The latter supplies some Icons and Logos, serving rather cosmetic aspects.

External Online Resources

There are many resources in the internet with in-depth coverage of most any C++ feature, at basic, intermediate, and advanced levels.

> This document does not duplicate information readily available elsewhere but rather provides links to it.

If you view the presentation on a computer anyway it is suggested to prefer the HTML-version over the PDF-version, as you may

- may navigate to related information by simply clicking the links, and
- may also view the presenter's notes and even choose to use them for your own annotations.

The PDF-version has the (small) advantage that it is a single file that can be used "stand-alone".*

*: The PDF version is most easily created from Google Chrome or some chrome-based browser with *Export to PDF* or *Print to PDF* or, if the final goal are print-outs anyway, by direct printing.

Online C++ References

For any non-trivial technical product, at some point you need to look-up detailed information, as e.g. it is provided in reference manuals.

Both now on many of their pages have example sections with has code that can be

- compiled as is, or
- modified and compiled again.

The former uses Coliru (= **Co**mpile, **li**nk, and **ru**n) the latter C++ shell as online compile service (see next page).

*: Besides the coverage of the standard library cppreference.com also has sections on language features. As these take a rather formal and dry approach, they are – of course – precise but often use phrasing that is not easy to comprehend, and is also rich in details which are unimportant for an initial understanding.

Online C++ Compilers

In recent years free *Online Compile Services* became available.*

> These are basically internet sites with web front-ends to one or more programming languages (compiled or interpreted).

C++ Online Compilers are great tools and while details vary many allow

- not only to try small programs (e.g. as "proof of concept")
  - without installing a compiler and library
  - let alone a whole development environment,
- but also to compare the behaviour of several compiler versions or brands easily, and
- to save such code examples returning a *permalink* which you then may send to others to discuss the code, be it
  - privately with colleagues and friends, or
  - in public on platforms like stack-overflow.

*: While some have been switched of again Coliru – which the author of this uses a lot – has been there since many years and still is. Wandbox is a more recent one with a little more embellished UI. With both you may choose between *g*++ and *clang*++ and both include a recent version of the Boost Library.

The C/C++ Preprocessor

The C preprocessor was introduced long ago to automate several kinds of systematic editing:*

- write common parts of several files only once (`#include`)
- conditionally compile portions of the source code (`#if`, `#ifdef`, ...)
- repeatedly insert (nearly) identical code sequences to avoid function call overhead for tiny subroutines (`#define`)

In C++ many "classic" use cases of the preprocessor were superseded by builtin language features, though at times it still comes in handy.

> But also keep in mind:
> The C Preprocessor doesn't know about C.
> ☺

*: Also, in the early years of C, on the PDP-11, separating some features from the compiler proper was necessary as the executable program file was limited to 64+64 kB (Code+Data).

Preprocessor Use Cases

The set of still valid use cases is small, but not empty:

- Centralising common code via Header Files.
- Using Include Guards to avoid header files being processed more than once.
- Special Usage Macros

An example where a macro is indispensable is this, where `assert` first evaluates its argument as C++ expression ...

```
void f(const char *cp) {
 assert(cp != nullptr);
 …
}
```

... but also requires it as character string for output, if the assertion fails.

- The key mechanism here is only provided for macro arguments (as part of macro replacement text) and called *Stringizing*, requested with # in front of a macro argument name.
- Similarly, ## is the so-called *Token Pasting* operator*, exclusively available in macro replacement text too.

*: A use case for token pasting is provided in the exploration section for Configurable Exceptions.

Header Files

Header files can be supplied by the system or be specific for a project:

- System header files are looked-up in a set of predefined locations.*
- Project specific header files are *first* looked up in locations associated with the project, *then* in the locations of the system header files.

Only regular slashes as path separators in include file names are portable (assuming the directory structure is matched), **backslashes are not**.

Example for system header files:

```
#include <cassert>
#include <iostream>
#include <string>
…
```

Example for project specific header files:

```
#include "mylib/utilities.h"
#include "point.h"
…
```

*: C++ implementations have the freedom instead of reading a file that exists in the file system to enable a predefined set of features within the compiler. Furthermore in C++ any system header **may** include other system headers. This may cause (mild) portability issues when switching compilers or even a complete development environment.

Include-Guards

Since the early times C, due to the

> ODR – One Definition Rule

it is a well-known "best practice to

- wrap all the actual content of a header file in a big conditional compilation,
- so that duplicating an `#include` for it does no harm.*

```
#ifndef MY_HEADER  // <--- begin of header file content
#define MY_HEADER
     // ^^^^^^^^^-- anything valid as unique macro name
     // …
     // … actual header content
     // …
#endif // <--------------- end of header file content
```

*: It is common practice to include one header from another header if the second depends on the first, though be aware of the potential problems caused by cycles in the dependency graph.

Special Usage Macros

The following macro is often useful for creating output* in tiny programs used for testing and demonstrations,

- not only **showing the value of some expression**,
- but also that very expression as text, after
- the name of the compiled file, and
- the line number from where the macro was called.

```
#define PX(expr)\
    (void)(std::cout << __FILE__ << '[' << __LINE__ << ']'\
                     << "\t" #expr " --> " << (expr) << std::endl)
```

*: Same as useful is a macro written for the purpose of showing a type:

```
#define PT(...)\
    (void)(std::cout << __FILE__ << '[' << __LINE__ << ']'\
                     << "\t" #__VA_ARGS__ " --> "\
                     << boost::typeindex::type_id_with_cvr<__VA_ARGS__>()\
                     << std::endl)
```

How C++ Extends C

- Arithmetic Types
- Const-Qualifier
- Pointers and Arrays
- References
- Default Arguments
- Function Overloading
- Operator Overloading

It should be understood that "under the hood", C and C++ are both mapped to the hardware in the same, basic way.

So, from the low-level view, they use the same Execution Model.

Arithmetic Types

C++ has all the arithmetic types that C has, including a special type for binary logic (`bool` with value `true` and `false`).

Conversions between arithmetic types take place automatically:*

```
int i = -12;       // always OK
unsigned u = i;    // OK (maybe compile time warning)
short t{32768};    // new C++11 style initialization,
                   // overflow error at compile time if
                   // short is 16 bit two-s complement
char c = t+122;    // OK (maybe compile time warning)
if (c) …           // every value not 0 is taken as true
```

C++11 allows to use `auto` as typename; the type of the variable will be the type of the initializing expression (without `const` and *reference*):

```
auto x = 3.1415;  // x will have type double
auto y = 1.4142f; // y will have type float
```

*: There may be a warning if loss of information is possible. This is typically based on the types involved, not on some analysis of actual values (which may or may not be fully determinable at compile time).

Brace Initialization

C++11 also extended the initialization syntax

- to allow
  - initial values be written in a pair of curly braces anywhere* and
  - (only in this case) omitting the equals sign between the variable name and the opening brace.
- provided a whole new (template) type `std::initializer_list<T>`
- written as comma-separated list of identical or compatible types
  - enclosed in a pair of curly braces.

> **[!]** While the original intent was to "simplify" the initialisation syntax, there were some unexpected stumbling-stones in combination with `auto`.

*: Traditionally C and C++ supported this syntax only for structures and native arrays.

Enforcing Conversions / Suppressing Warnings

Conversions need sometimes to be enforced with cast operations:*

```
int x = 9;
int y = 5;
double d = static_cast<double>(x) / static_cast<double>(y);
        // instead of C-style cast: (double)x / (double)y
```

Also, cast operations are sometimes used with the intent to suppress a warning:

```
short s = static_cast<short>(d);
```

> Which warning is issued under what circumstance and how it can be suppressed is always compiler specific.

*: C-style casts are still part of the C++ language but a deprecated language feature because they are hard to spot and the intention is not as much part of the syntax as with the C++-style cast, for which more examples will be shown later.

The const Qualifier

A variable may optionally be qualified with const:*

```
const int x = 17;
double const PI = 3.14;
```

The effect is that

- there must be an initialisation and
- subsequent attempts to modify the variable result in a compile time error.

As the examples show, the const qualifier may be written to the left or to the right of the type to which it applies.

*: The const qualifier was originally introduced with C++ but also became part of the C89 standard – hence today, 25 years later, it may well be reclaimed as a C++ addition but few people still know. Nevertheless it seems even today C++ programs make more use of this feature than C programs.

Arrays and Pointers

Arrays and pointers are (sometimes) confused because for function arguments there are two different ways to express the same concept:

```
void foo(short arr[]) {
    …
}
```

```
void bar(short *p) {
    …
}
```

Both are essentially equal as – behind the scenes – the functions will be called with the address of an `int` (arithmetic type).

The difference in information conveyed in the two cases is rather about intent:

- `foo` will (presumably) be called with the address of the first element of an array of `int`-s;
- `bar` will be called with the address of a (presumably) single `int` or maybe even a null pointer.

Indexing and Address Arithmetic

When used to define some data item, arrays and pointers are different:

For an array storage space is set aside to store the requested number of items of the given type:

```
double data[20];
char message[1024];
```

Individual items are typically accessed by giving their index:

```
… data[12] …
… message[i] …
```

Given a pointer points to an array cell …

```
dp = &data[0];
cptr = &message[i];
```

For a pointer only memory space to hold an address is set aside:

```
double *dp;
char *cptr;
```

Assuming an appropriate initialisation items pointed to are accessed by dereferencing:

```
… *dp …
… *cptr …
```

… also address arithmetic can make sense:

```
… ++dp …
… *(cptr-3) …
```

Equivalence of Indexing and Address Arithmetic

There are two basic rules that make pointers and arrays equivalent when it comes to indexing and address arithmetic:

- In almost any context* the pure name of an array decays to a pointer to the first array element, i.e. `data` and `&data[0]` are equivalent.
- All pointer arithmetic is scaled with the storage size of the underlying type, making
  - `ptr+i` equivalent to `&ptr[i]` and
  - `*(ptr+i)` equivalent to `ptr[i]`.

In C++ the need to use C low-level data types like arrays and pointers strongly diminishes as the standard library offers more convenient abstractions like Container Classes and Smart Pointers.

*: Exceptions are when the name of an array is used with the `sizeof` of `&`-operator: then it will keep its array type, so `sizeof data` is the size the whole array occupies (in bytes) and `&data` is the address of the whole array, which is – of course – the same as the address of its first element but will use a different scaling for address arithmetic.

Arrays

Arrays belong to the C++ subset of language features assumed from C:

- the size must be fixed at compile time.
- unless an initialisation is supplied:

```
double data[200];
int primes_table[] = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29
};
```

In addition to the array data there is often some extra information stored (in separate variables) up to which element an array is filled …

```
double *filled = &data[0]; // location of next free space
*filled++ = …; // idiomatic style to fill array with values
```

or how many entries exist in an initialised array:

```
const auto N = sizeof primes_table / sizeof primes_table[0];
```

Arrays (2)

The size information needs eventually be presented to functions that process the content of same array, as the pure array name – when used as function argument – decays to a pointer to the first array element:*

```
void print_primes(int table[], std::size_t count) {
    using namespace std;
    for (int i = 0; i < count; ++i) cout << table[i] << endl;
}
…
print_primes(primes_table, N);
```

Or:

```
double sum_data(double *from, double *to) {
    double s = 0; while (from < to) s += *from++; return s;
}
…
auto sum = sum_data(data, filled);
```

*: The C++ Standard Library provides the classes `std::array` and `std::vector` that may often be used as drop-in replacements for built-in arrays but bundle size information with the data.

`const` Arrays

Arrays qualified with `const` must be initialised and cannot be modified:

```
const int primes_table[] = {
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29
};
```

> Initialised write protected arrays are good for pre-calculated tables of data as it can be loaded into initialised storage from the program's executable image.

As `const` in C++ silently implies internal linkage, to share such data between translation units its declaration must also be qualified with `extern` in a header file included by all interested parties:

```
extern const int primes_table[];
```

The initialisation from above then goes into **exactly one** translation unit.

---

*: The rule *`const` implies internal linkage* allows to put the definition and initialisation of `const` data into header files without multiple definitions of the same symbol leading to linker errors.

Pointers

Pointers belong to the C++ subset of language features assumed from C.

Differently typed pointers are generally incompatible with each other:

```
int i;   // a memory location to hold int-s
int *ip = &i; // a pointer to that location
float *fp = ip; // ERROR
```

If the above assignment to `fp` would pass, via `*fp` the memory location holding an `int` could be accessed as if it were a `float`.*

This also defeats usage errors of multi-level pointers:

```
int **ipp = &ip;  // pointer to pointer
int x = 3 * *ipp; // single indirection still leaves pointer
                  // which cannot be multiplied by an int
```

*: Write access could even extend beyond that and change unrelated variables, if the type that determines memory space is smaller that the type the pointer claims to point to.

Typed Pointers vs. Generic Pointers (`void *`)

There is a difference between C and C++ in compatibility of typed and generic pointers in, which adds more type safety:

In C type compatibility is both ways, from generic to typed and from typed to generic pointers:

```
int *ip;
…
void *p = ip;  // C: OK
float *fp = p; // C: OK
```

In C++ type compatibility is one-way – from generic to typed is an error:*

```
int *ip;
…
void *p = ip;  // C++: OK
float *fp = p; // C++: ERROR
```

When at a given point the true nature of a generic pointer is known, it may be assigned back to the original typed pointer type using a cast:

```
int *ip2 = static_cast<int*>(p);
```

*: Despite slightly improved type safety in C++ generic pointers cut a big hole in compile-time type checking, as anyway at some point there needs to be a cast which requires that type information is somehow embedded in the program logic. If the set of possible types is bounded at design time, class hierarchies are usually a superior alternative while for type-generic coding – e.g. in basic libraries – parametrizing types via templates provides stringent type-checking at compile-time.

Arbitrary Type Conversions

With the possibility of arbitrary type conversions it is easy to introduce type errors, in C++ too.

While `static_cast` requires a generic pointers as intermediate, there is even a stronger tool (say sledge-hammer?) which directly fits together what really shouldn't fit:

```
int *ip;
…
float *fp = reinterpret_cast<float*>(ip);
```

Using pointer type conversions with `static_cast` and especially using any conversion with `reinterpret_cast` means **a lot of responsibility** is assumed for what is done is actually correct.

> As C++ has far better language support for generic programming as C, low-level cast operations can be avoided in most any case.*

*: One of the few legitimate uses of `reinterpret_cast` is in driver programming to specify hardware addresses for memory mapped devices.

`const` Qualifier for Pointers

Be sure to understand how `const` applies if pointers are involved!

The pointer itself is constant (and hence needs to be initialised):

```
T *const p = …;
```

The memory location reachable via the pointer is constant:

```
T const *p;
```

Same as:

```
const T *p;
```

In some situations even both should be protected against modifications, the pointer and the memory to which it points:*

```
const T *const p = …; // a unmodifiable pointer to an
                      // unmodifiable memory location
                      // of type T
```

*: Of course `T const *const p = …;` would have the same meaning.

References

References are more or less a different syntax for pointers.

- They are declared with a leading `&` – instead of a leading `*`;
- They must be initialised with some variable to which it refers,
  - i.e. the `&`-operator – take address – is implicitly applied to that variable.
- When used, a reference is automatically dereferenced,
  - i.e. the `*`-operator – follow pointer, get content – is implicitly applied.

> Conceptually a reference is always connected to an existing entity – there is no such thing as a "Null Reference" akin to a "Null Pointer".*

*: Nevertheless, an incautious programmer may accidentally create invalid references, e.g. like that: `void f(T *p) { …; T &r = *p; … }` (when `p` is not checked against the Null Pointer)

References Viewed as Aliases

A reference may also be viewed as an alias for an existing variable:

```
int i;
…
int &r = i;   // any access to r effectively accesses i
```

The main use for this feature is for function arguments, where it minimises the syntactical overhead of explicit pointer use.

References versus Pointers

Two function that swap their argument values, one using pointers, one using references ...

```
void swap(int *p1, int *p2)
{
    int tmp = *p1;
    *p1 = *p2;
    *p2 = tmp;
}
```

```
void swap(int &r1, int &r2)
{
    int tmp = r1;
    r1 = r2;
    r2 = tmp;
}
```

... and their usage:

```
int a, b;
…
if (a < b)
    swap(&a, &b);
```

```
int a, b;
…
if (a < b)
    swap(a, b);
```

`const`-References

As references cannot be changed after initialization* they really are like `const` pointers – but this should not be confused with pointers to `const`:

```
const int ci = 42;
int i;
int &ri = i;          // OK
const int &cri = ci; // OK
const int &cri2 = i; // OK
ri = …;    // OK to modify
cri2 = …;          // ERROR
int &ri2 = ci;     // ERROR
```

```
const int ci = 42;
int i;
int *const ip = &i;
const int *const cip = &ci;
const int *const cip2 = &i;
*ip = …;   // OK to modify
*cip2 = …;         // ERROR
int *ip2 = &ci;    // ERROR
```

Any access to `ri`, `cri`, and `cri2` actually accesses the referenced variable, while applying `&` takes the address of it. Vice versa with pointers: here the plain variable usage represents the address held and applying the dereferencing operator `*` accesses the content of that memory location.

*: To put it more clearly: a reference refers to the same storage location all of its lifetime, like a pointer that cannot be set to a different location. A different question is whether the storage location *reachable* through the reference can be modified – and this is what this page is about.

Lvalue- and Rvalue-References

A special property of `const` references is that they may be initialised with temporaries, typically resulting from an expression of matching type:*

```
int a, b;
…
int &r = a;             // OK
int &r2 = a + b;        // ERROR
const int &r3 = a + b; // OK
```

In the last case temporary memory space must be created – for the result of the addition – and will be kept alive for the life-time of the reference.

A recent addition of C++11 are Rvalue References.
These can only be initialised with temporaries.

```
int &&rr = a + b; // OK
int &&rr2 = a;    // ERROR
int &&rr3 = a + 0;   // OK
```

The main use of rvalue-references is to add move-versions of (copy-) constructor and assignment, which will be covered later.

*: Early versions of C++ allowed this for non-`const` references too but it turned out to be an error prone feature and was finally removed in later versions of C++ on which C++-98 was based.

When to Use References?

References should be generally preferred as function arguments for

- handing over anything else besides basic types efficiently, as only an address needs to be transferred, not all of the data structure copied;
- giving access to the object itself for applying modifications.

> Use a `const` qualified reference to protect some function argument from modifications while – behind the scenes – only handing-over its address.

ADT* to implement a counter that restarts at a given value once it had reached a given limit:

```
typedef struct {
    int limit; // maximum value
    int start; // restart value
    int ticks; // current value
} counter;
```

```
int value(const counter &c) {
    return c.ticks;
}
```

```
void count(counter &c) {
    if (++c.ticks > c.limit)
        c.ticks = c.start;
}
```

*: ADT means abstract data type and (in the example above) comes close to what can be done with classes in C++. (In a "pure C" solution references can be easily replaced by pointers.)

Default Arguments

Function arguments can have defaults, which means that the actual call can leave out such arguments.*

```
void say_hello(const char *greet = "hello") {
    cout << greet << endl;
}
```

Possible calls:

```
say_hello();
say_hello("salve");
```

The default value must be present in a prototype that declares the function and in this case the argument name is optional:

```
extern void say_hello(const char * = "hello");
```

Function Overloading

Functions may be overloaded based on their signature, which consists of

- the function name and argument list
- but **not** the function return type.

```
void say_hello(const char *greet) {
    cout << greet << endl;
}
void say_hello(int repeat, const char *greet) {
    do
        cout << greet << endl;
    while (--repeat > 0)
}
```

Possible calls:

```
say_hello("hi!");
say_hello(3, "hi!");
```

Operator Overloading

Operators may be overloaded too, but only if at least one of operand is not a basic type.*

```
enum Color { Red, Blue, Green };
Color operator++(Color& c) {
    switch (c) {
        case Red: return c = Blue;
        case Blue: return c = Green;
        case Green: return c = Red;
        default: assert(!"never executed");
    }
}
```

*: For basic types operators are already defined – or forbidden – and that meaning cannot be changed.

Object Oriented Programming

This chapter introduces not a minor but a substantial extensions C++ has over C, using a "limit counter" implemented as abstract data type as an example that gets gradually extended.*

- From `struct` to `class`
- Access Protection
- Constructors
- Base and Derived Classes
- `virtual` Member Functions

> At the low-level C++ classes still are data `struct`-ures mapped to memory, while *Composition* and *Inheritance* are more or less nested data structures with some specific rules for type (in-) compatibility.

*: There are generally two roads leading to C++, the first one, which is taken here, *step by step* turns C style programming into making use of classes and objects. The other road would radically cut off with all "*traditionally thinking*" and especially all "*old code*" and mandate a "*completely different approach*" from "*classic procedural design*". It will often start with teaching design methodology, probably via the use of UML Diagrams. Though some proficiency with the latter will surely pay to document OOP High-Level Designs it is **not at all** necessary to get substantial added value by a move from C to C++.

From `struct` to `class`

Turning Functions into Members

C++ allows to define functions that take a pointer to a `struct` as (first) argument inside the class as members (aka. member functions):

```
struct LimitCounter {
    int start;
    int limit;
    int ticks;
    void init(int st, int lim);
    void count();
    void reset();
};
```

The argument to hand over the limit counter data is now implied and not named any more in the list of arguments.

Defining `LimitCounter` Objects

Objects (aka instances) of limit counters would be defined as follows:

```
LimitCounter lc1, lc2;
```

Calling Member Functions

When calling the member function syntactically the object is on the left:

```
lc1.init(1, 10);
lc2.init(0, 59);
```

From a low-level point of view there is no difference to a global function with an explicit pointer argument, as the address of the objects `lc1` and `lc2` is received by the member function as `this` (-pointer).

Implementing Member Functions

Of course, the function inside the class are just declarations that must be implemented too. This would look like follows:

```
void LimitCounter::init(int st, int lim) {
    this->start = this->ticks = st;
    this->limit = lim;
}
```

As `this->` is assumed by default for any name that is not local or part of the argument list, it is usually omitted:

```
void LimitCounter::count() {
    if (++ticks > limit)
        ticks = start;
}
```

Implementing Member Functions Inline

For efficiency reasons it usually makes sense to define small member functions wtih the `inline` keyword so that their body code is inserted instead of subroutine calls at the point of their use:

```
inline
void LimitCounter::reset() {
    ticks = start;
}
```

Then a "call" to this (simple) function like `lc1.reset()` will typically result in a single `mov` instruction at the machine level, which is both, faster **and** smaller as a subroutine call!

Implementing Members Inside the Class

Furthermore `inline` is automatically assumed if the function is implemented inside the class to which it belongs:

```
struct LimitCounter {
    …
    void reset() {
        ticks = start;
    }
    …
};
```

Whether to use this or an explicit `inline` with an implementation outside of the class is largely a matter of taste.

When classes are – as usual – defined in separate header files, `inline` implementations **must go to the header file** because the compiler needs to know them when it emits the code for a "function call".

Access Protection

It is usually considered "good style" if a user defined type protects its data members from direct access but adds accessors instead:

```
class LimitCounter {
    int start;
    int limit;
    int ticks;
public:
    …
    int value() const {
        return ticks;
    }
    void count() {
        if (++ticks > limit)
            ticks = start;
    }
};
```

The `const` qualifier for the member function – between the closing parenthesis and the opening curly brace – allows to call this function for non-modifiable `LimitCounter` instances:

```
void
foo(const LimitCounter &c) {
    cout << c.value(); // OK
    c.count();      // ERROR
}
```

*: There are several common naming schemes for such accessors. A project should probably chose one of this schemes and apply it consistently for all its classes.

Levels of Access Protection

There are several protection levels which can be interspersed as often as necessary with the members in `struct`- or `class`-blocks like labels:*

- `public:` – accessible from outside
- `private:` – accessible for member functions only
- `protected:` – accessible for member functions and derived classes (covered later)

```
class Mine {
public:
    …
protected:
    …
private:
    …
};
```

```
class Other {
    …
public:
    …
private:
    …
public:
    …
};
```

*: Despite the similar syntax these are **not** labels, as the names are reserved keywords and the syntax limits their use to `struct` and `class` blocks.

`struct` vs. `class`

The only difference between `class` and `struct` is that the latter starts by default with a `public:` section and the former with a `private:` section.

```
struct Mine {
private:
    …
public:
    …
};

struct Other {
    …
private:
    …
};
```

```
class Mine {
    …
public:
    …
};

class Other {
public:
    …
private:
    …
};
```

Constructors

Classes may have Constructors which will – if they exist – be called automatically when an instance of the given class comes into existence:

```
class LimitCounter {
    …
public:
    LimitCounter(int st, int lim) {
        start = ticks = st;
        limit = lim;
    }
};
```

Constructor arguments must now be supplied with the definition of instances:

```
    LimitCounter lc1(1, 10), lc2(0, 59);
```

Constructors: Member Initialisation Lists

There is an alternative syntax to initialise data members, the *MI-list* (short for member initialisation list). Its use is mandatory when constants and references are initialised, but they may optionally be used always:

```
class LimitCounter {
    const int start;
    const int limit;
    int ticks;
public:
    LimitCounter(int st, int lim)
        : start(st), limit(lim), ticks(st)
    {}
    …
}
```

It is not at all unusual that – when the MI-list is used for all members – the constructor body stays empty.*

*: Of course, to maximize consistency by re-use, it might alternatively be considered in the above example to initialise `ticks` with a call to `reset()` from inside the constructor body.

Destructors

The destructor is the complementary operation to the constructor.

It has the same name as the class prefixed with ~ and will automatically be called if an instance of the given class ceases to exist.

There is no need for a destructor in the `LimitCounter` class.

> An example for a class that actually needs a destructor follows in the section on Dynamic Memory Allocation.

Static Members

Member variables and functions may be ´static`, which means they exist only once (for the class), not for every instance.

As there is no need for static members in the `LimitCounter` example, the following shows how a class could count the number of its instances:*

In the header defining the class:

```
// file: Foo.h
class Foo {
    …
    static int instances;
public:
    Foo() { ++instances; }
    ~Foo() { --instances; }
    …
};
```

In exactly one translation unit:

```
// file: Foo.cpp
…
int Foo::instances = 0;
…
```

*: This example is actually incomplete as it omits the copy constructor (not covered so far) which would also need to be implemented as it must increment the `instances` count too.

Base and Derived Classes

Classes may be derived from other classes if they only want to extend some existing functionality.

The following defines a new class `OverflowCounter` which adds to a `LimitCounter` the capability that in case of an overflow it increments a connected counter.

The approach is shown step by step on the next pages.

Derive from a Base Class

The syntax names the base class after the derived:

```
class OverflowCounter : public LimitCounter {
    …
};
```

The use of `public` here implies that an `OverflowCounter` can always replace a `LimitCounter`, which will become important soon.

> If this kind of substitutability is not desirable, a `private` base class should be used.

Add Members

A derived class is always free to add member variables and functions* to what is inherited from its base class.

> The `OverflowCounter` needs to add a reference to the next stage (which will count on overflow).

Note that this is a different instance than the `LimitCounter` from which the `OverflowCounter` is derived!

```
class OverflowCounter : public LimitCounter {
    LimitCounter &next;
    …
};
```

*: In this example no member functions are added.

Add A Constructor

To initialise data members added, a derived class usually needs its own constructor.

```
class OverflowCounter : public LimitCounter {
    LimitCounter &next;
public:
    OverflowCounter(int st, int lim, LimitCounter &n)
        : LimitCounter(st, lim), next(n)
    {}
    …
};
```

The constructor forwards its first two arguments to the constructors of its `LimitCounter` base class and uses the third (reference) argument to initialise `next`.

OverflowCounter Overwriting `count`

Of course, an `OverflowCounter` has a slightly different way to count compared to a `LimitCounter`.

> Therefore it needs to replace the inherited `count()` member function.*

```
void OverflowCounter::count() { // overriding inherited count
    int old = value();     // calling inherited value()
    LimitCounter::count(); // calling inherited count()
                           // (qualification necessary here
                           // to avoid recursive call !!)
    if (value() < old)
        next.count();  // calling next stage `count()`
}
```

While this solution works, it looks (and is) clumsy, but any improvement would mean that the base class needs to take into account the needs of its derived class.

---

*: The technical term here is "overriding", though in German literature this is often translated with "überscheiben" (= overwriting).

Reducing Protection of Members

One possibility for a more elegant way would be a reduced protection of its `limit` data member. The following makes it accessible for a derived class:

```
class LimitCounter {
    …
protected:
    const int limit;
    …
}
```

The solution possible with this modification still looks clumsy:

```
void OverflowCounter::count() {
    const bool will_reset = (value() == limit);
    LimitCounter::count();
    if (will_reset)
        next.count();
}
```

Giving Hints to From Base Classes

Another possibility were to have the base class give a hint to the derived class.

In the scenario used as example so far this hint could be the return value of the `count` member function:

```
bool LimitCounter::count() {
    if (++ticks > limit) {
        ticks = start;
        return true;
    }
    return false;
}
```

Using Hints in Derived Classes

The derived class would now make use of this hint ...

```
bool OverflowCounter::count() {
    if (LimitCounter::count()) {
        next.count();
        return true;
    }
    return false;
}
```

... but assuming it may in turn be used as base class from some derived class that needs this hint, it has to forward it accordingly via its own return value.

Still not a really nice solution!

`virtual` Member Functions

As a special flavour of

- General *Runtime Type Identification*
- or RTTI* in short

C++ provides virtual member functions.

> It should be understood that **calling** a `virtual` Member function comes at the cost of some indirection.

Furthermore, in most cases the body of a `virtual` member function can not be expanded inline, leading to a substantial performance penalty on modern hardware with fast local instruction caching and pre-fetch mechanisms, especially for small functions which otherwise are often desirable for modularity and other reasons.

---

*: Another examples for the practical use of virtual member functions follows in a later section. Two very specific examples are in the optional Info-Graphics here – comparing the implicit and the explicit approach to run-time type identification – and here – comparing two approaches in the spirit of the *Template Method Pattern*. (In great extent these examples can be found in books dealing extensively with OOP *Design Patterns*, like the GoF-Book).

Adding Extension Points in the Base Class

A far superior solution is possible if the base class (`LimitCounter`) had anticipated the needs of its derived classes and added an extension point:

```
class LimitCounter {
    …
    virtual void overflow() {}
public:
    void count();
    …
};
```

The base class does nothing in this case – except calling a function that returns immediately:

```
void LimitCounter::count() {
    if (++ticks > limit) {
        ticks = start;
        overflow();
    }
};
```

Hooking Code to Extension Points from Derived Classes

This would be the **full** code of the class `OverflowCounter` then:

```
class OverflowCounter : public LimitCounter {
    LimitCounter &next;
    virtual void overflow() {
        next.count();
    }
public:
    OverflowCounter(int st, int lim, LimitCounter &n)
        : LimitCounter(st, lim), next(n)
    {}
};
```

Especially there is no need any more to override the `count()` member function inherited from `LimitCounter` – instead the function for the extension point is overridden one.*

*: An advantage achieved thereby is that it cannot be forgotten to call the inherited function from the overwritten one.

A Chain of Counters

Now a chain of counters can be build, e.g. to act as display of a clock:

```
int main() {
    LimitCounter hours(0, 23);
    OverflowCounter minutes(0, 59, hours);
    OverflowCounter seconds(0, 59, minutes);
    …
    // probably in a loop ...
        seconds.count();
    …
}
```

The fact that the "middle" counter (`minutes`) can be used as constructor argument for another `OverflowCounter` is worth mentioning here:

> It is only possible because the `OverflowCounter` has the `LimitCounter` as `public` base class and therefore – as an instance of the derived class – the former is always accepted as a substitute for the latter.

Alternative Design

An alternative design could use **one class only** that is similar to the `OverflowCounter` used so far. The difference is that it connects the next counter – the one which counts on overflow – via the member `pnext` which is a pointer and is a null pointer if there is no next counter attached.

```
class OverflowCounter {
    const int start;
    const int limit;
    int ticks;
    OverflowCounter *pnext;
public:
    OverflowCounter(int st, int lim, OverflowCounter *pn = 0)
        : start(st), limit(lim), pnext(pn)
    {}
    void count() {
        if ((++ticks > limit) && (pnext != 0))
            pnext->count();
    }
    …
};
```

Introduction to Exceptions

Basically exceptions are

- non-local branches,
- backwards in the call tree,
- to a `catch`-block in a function that has not yet returned,
- and its preceding `try`-block active.*

*: Some more specific information on exceptions is provided in an separate Info-Graphic.

Throwing Exceptions

A `throw` statement is used to terminate current processing.

It needs to be followed by an expression, which has a type and a value.*

- Sometimes basic types are used as exceptions,
- but typically a class (or small class hierarchy) is defined for that purpose.

```
… throw 42;                 // integer thrown
… throw "let me out here!"; // const char* thrown
… throw ErrorX(12, 15);     // instance of class ErrorX thrown
```

*: A `throw`-statement usually is only conditionally, i.e. when a certain condition is detected that prevents following the ordinary flow of control any further.

Catching Exceptions

The `catch` block defines a parameter which acts much like a function argument.*

- Actually it receives the value of the expression thrown,
- therefore its type must be compatible with the type thrown.

Catch 42 (and other int-s):

```
try { … } catch (int x) { … }
```

Catch `"let me out here"` (and other C-style strings):

```
try { … } catch (const char* txt) { … }
```

Catch instances of class `ErrorX` and classes publicly derived from `ErrorX`:

```
try { … } catch (ErrorX &ex) { … }
```

*: The examples shown here are written into a single line only to safe some space. Usually the `try`- and `catch`-block would be formatted to extend over several lines, as is necessary from its contents. The examples shown here are written into a single line only to safe some space. Usually the `try`- and `catch`-block would be formatted to extend over several lines, as is necessary from its contents.

Configurable Exceptions

A strategy to make the throwing of exceptions configurable could be not to throw directly but through a virtual member function:*

```
struct ErrorX {
    ErrorX(int, int);
};
…
class Mine {
    virtual void error_X(int, int) = 0;
public:
    void do_it() {
        …
        if (…) error_X(12, 15);
        …
    }
};
```

*: As shown so far `Mine` is an abstract class which can only be used as base class for derived classes like those shown on the next pages. Of course one of these following classes could also serve as default implementation.

Configurable Exceptions (2)

Instances of the following class actually throw when the error happens:

```
class ThrowingMine : public Mine {
    virtual void error_X(int a, int b) { throw ErrorX(a, b); }
}
```

Instances of this class ignore the problem:

```
class NonThrowingMine : public Mine {
    virtual void error_X(int, int) { /*empty*/ }
}
```

Dynamic Memory Allocation

Dynamic memory (aka *heap allocation*) is requested with:

- `new T` – where `T` is any datatype, memory has sufficient size to store and proper alignment to store one item of type `T`;
- `new T[n]` – where T is any datatype and `n` is a positive integral number, memory has sufficient size ans proper alignment to store `n` items of type `T` as they were stored in an array.

Dynamic memory is release with:

- `delete p` – where `p` is of type pointer to `T` and has received its value from `new T`;
- `delete[] p` – as before but `p` has received its value from `new T[n]`.

[!] Releasing memory that was allocated with `new T[n]` with `delete` or releasing memory that was allocated with `new T` with `delete[]` results in undefined behavior.

`RingBuffer` Example

A class RingBuffer to store `N` integers for easy FIFO-access might have the following implementation:

```
class RingBuffer {
    int alloc;    // ORDER DEPENDENCY (1)
    int *data;    // ORDER DEPENDENCY (2)
    int iput;
    int iget;
    static int wrap(int x) { return x % alloc; }
public:
    RingBuffer(int n);
    ~RingBuffer();
    bool get(int &);
    void put(int);
    bool empty() const { return iput == iget; }
    bool full() const { return wrap(iput + 1) == iget; }
    RingBuffer(const RingBuffer &) = delete;
    RingBuffer &operator=(const RingBuffer &) = delete;
};
```

(The ORDER DEPENDENCY comment will be explained on the next page.)

`RingBuffer` Example (2)

Memory management is done from the constructor and destructor:

```
// constructor allocating dynamic memory
RingBuffer::RingBuffer(int n)
    : alloc(n+1), data(new int[alloc]), iget(0), iput(0);
{}

// destructor releasing dynamic memory
RingBuffer::~RingBuffer() {
    delete[] data;
}
```

> **[!]** From looking into the MI-list it seems obvious that `alloc` is initialised prior to its use in the initialisation of `data` ...

... but initialisation order is determined by the order of member variables in the class definition – changing this order may break the above code.*

*: DEPENDENCY ORDER comments in the class faithfully try to defeat this but it might be wiser to chose a defensive and more robust style, e.g. by using `n+1` instead of `alloc` in the initialisation of `data`.

`RingBuffer` Example (3)

By deleting the copy constructor and the assignment operator an object of type `RingBuffer` may not be copied or assigned.

If the `RingBuffer` should be copyable, the implementation of the copy constructor could look as follows:

```
RingBuffer::RingBuffer(const RingBuffer &init)
    : alloc(init.alloc), data(new int[alloc])
    , iget(init.iget), iput(init.iput) {
    std::memcpy(data, init.data, alloc*sizeof(int));
}
```

It would be called in scenarios like this:

```
void foo(RingBuffer arg); // call by value argument!
RingBuffer a(10);
…
RingBuffer b = a; // copy constructor initialises new object
RingBuffer c(a);  // b as before, only different syntax
foo(b); // copy constructor used to initialise value argument
```

`RingBuffer` Example (4)

The implementation of copy assignment operator could look as follows:

```
RingBuffer& RingBuffer::operator=(const RingBuffer &rhs) {
    if (this != &rhs) {
        alloc = rhs.alloc;
        int *tmp = new int[alloc];
        std::memcpy(tmp, rhs.data, alloc*sizeof(int));
        delete[] data;
        data = tmp;
        iget = rhs.iget;
        iput = rhs.iput;
    }
    return *this;
}
```

It would be called in scenarios like this:

```
RingBuffer a(10), b(100);
…
b = a; // copy assignment discards memory space previously
       // allocated, allocates new space and copies content
```

`RingBuffer` Example (5)

If the `RingBuffer` needs only to be movable (or as an optimization if it is copyable and made movable in addition), the move constructor could look as follows:

```
RingBuffer::RingBuffer(RingBuffer &&init)
    : alloc(init.alloc), data(init.data)
    , iget(init.iget), iput(init.iput) {
    init.data = nullptr;
}
```

It would be called in scenarios like this:

```
RingBuffer foo();
void bar(RingBuffer);
…
RingBuffer a = foo(); // initialization with temporary
RingBuffer a(foo());  // as before, only different syntax
bar(foo());  // value argument initialised with temporary
```

`RingBuffer` Example (6)

Furthermore, the implementation of a move assignment operator could look as follows:

```
RingBuffer& RingBuffer::operator=(RingBuffer &&rhs) {
    alloc = rhs.alloc;
    delete[] data;
    data = rhs.data;
    rhs.data = nullptr;
    iget = rhs.iget;
    iput = rhs.iput;
    return *this;
}
```

It would be called in scenarios like this:

```
RingBuffer foo();
…
RingBuffer a(10);
a = foo(); // move assignment discards memory space previously
           // allocated, then "steals" pointer from temporary
```

`RingBuffer` Example (7)

Only for completeness: this are the member functions that fill in or take out elements:

```
// add an element
void RingBuffer::put(int e) {
    if (full()) iget = wrap(iget) + 1;
    data[iput] = e;
    iput = wrap(iput+1);
}
// retrieve an element
bool RingBuffer::get(int &e) {
    if (empty()) return false;
    e = data[iget];
    iget = wrap(iget+1);
    return true;
}
```

Pointers Revisited

One of the problem with pointers that their type does reflect too little of the use pattern.

- There are pointers that allow to access some memory that
  - exists already before the referring pointer comes into live
  - and will continue to exist when the life of the referrer ends.
- There are pointers that have exclusive ownership to dynamic memory
  - they need to `delete` when they go out of scope,
  - except they transfer ownership to some other exclusive owner.
- There are pointers that have non-exclusive ownership to dynamic memory that needs to be `delete`-ed when the life of the last of them ends.

Smart Pointers

C++11 introduced new types of pointers designed to solve the problem of unique and shared ownership:

- `unique_ptr<T>` (where `T` is any concrete type) implement the concept of unique ownership and therefore are not copyable but can be moved.
- `shared_ptr<T>` (where `T` is any concrete type) implement the concept of shared ownership maintaining a reference count.

Smart Pointers are provided as classes with overloaded pointer-operations to make them usable in an easy way – **just like native pointers**.

> Knowing the principles and some intricacies of their implementation **is NOT AT ALL a requirement** but can surely help to come to a better understanding and also some guidelines for their correct use.

`RingBuffer` Example with `std::unique_ptr`

If the `RingBuffer` used as example in the previous section has no requirement to be copyable it could be simplified as follows:

```
class RingBuffer {
    int alloc;
    std::unique_ptr<int[]> data;
    …
public:
    RingBuffer(int n)
        : alloc(n+1), data(new int[alloc]), …
    …
};
```

- No explicit move constructor and assignment is necessary any more as the default versions do the right thing.
- A copy constructor and assignment – doing the **wrong** thing by default – is **not** automatically supplied as `unique_ptr` is not copyable.

Beyond Unique Ownership

Besides `std::unique_ptr` C++11 added* two more "Smart Pointers":

- `std::shared_ptr` and
- `std::weak_ptr`

> Neither of both is covered here in detail, but feel free to ask for more information based on the Infographic "Smart Pointers" in the appendix.

In short, both apply in scenarios which runtime systems of other programming languages solve with Garbage Collection (or GC).

The main difference from true GC is that

- while consequently using `std::shared_ptr` solves the problem of "pointers going stale" (aka. "dangling references)
- in some circumstances there is still the need for a careful design mixing-in `std::weak_ptr` to reliably avoid memory leaks.

---

*: C++11 also deprecated `std::auto_ptr`, which – to quote Bjarne Stroustrup – should always have been what `std::unique_ptr` now is but couldn't, as C++98 lacked the distinction between *Move* and *Copy*.

Template Introduction

Templates are available as mechanism for classes and functions to parametrize types and compile time constants.

> Templates **avoid source code duplication*** when
> "***The Only Thing that Differs Is a Type***"
> (or some compile time constant)

An even more advanced view of templates is that they are

- Compile-Time Functions doing
- Type Transformations

finally leading to Template Meta-Programming

*: Emphasis here is on **source** code: As the "same" operations with different types typically require different machine instructions, the total amount of machine code is not different from a "Copy & Paste" approach with subsequent systematic changes.

Template Example

A `RingBuffer` like the one used in the last series of examples might also be useful for element types other than `int`.

```
template<typename T>
class RingBuffer {
    int alloc;
    T *data;
    int iput;
    int iget;
    static int wrap(int x) { return x % alloc; }
public:
    RingBuffer(int n);
    ~RingBuffer();
    bool get(T &);
    void put(const T&);
    …
};
```

Note that not **all** occurrences of the type `int` have been replaced, as some are not related to the element type but used as sizes or indices!

Template Example (2)

The constructor will change as follows:

```
template<typename T>
RingBuffer<T>::RingBuffer(int n)
    : alloc(n+1)
    , data(new T[n+1]),
    , iget(0)
    , iput(0)
{}
```

> Of course any function implemented outside the class – not only the constructor – must be modified accordingly.

Two more examples follow on the next page.

Template Example (3)

```
template<typename T>
void RingBuffer<T>::put(const T& e) {
    if (full()) iget = wrap(iget) + 1;
    data[iput] = e;
    iput = wrap(iput+1);
}


template<typename T>
bool RingBuffer<T>::get(T &e) {
    if (empty()) return false;
    e = data[iget];
    iget = wrap(iget+1);
    return true;
}
```

Template Example (4)

If sizing the `RingBuffer` dynamically is not required, the data array could be turned into a member variable and its size be given with another template argument:

```
template<typename T, std::size_t N>
class RingBuffer {
    T data[N+1];
    int iput;
    int iget;
    static int wrap(int x) { return x % (N+1); }
public:
    bool get(T &);
    void put(T);
    bool empty() const { return iput == iget; }
    bool full() const { return wrap(iput + 1) == iget; }
};
```

It should be noted that this change relieves the `RingBuffer` from the memory management burden it previously had.

Template Example (5)

The second argument now becomes also part of the member function definitions outside the class:

```
template<typename T, std::size_t N>
void RingBuffer<T, N>::put(T e) {
    if (full()) iget = wrap(iget) + 1;
    data[iput] = e;
    iput = wrap(iput+1);
}


template<typename T, std::size_t N>
bool RingBuffer<T, N>::get(T &e) {
    if (empty()) return false;
    e = data[iget];
    iget = wrap(iget+1);
    return true;
}
```

Though the parameter `N` was not used in the member function implementations shown so far, it is available, if necessary.

The following member function determines the count of elements stored:*

```
template<typename T, std::size_t N>
std::size_t RingBuffer<T, N>::size() const {
    return (iget < iput)
        ? (iput – iget)
        : (iput + N + 1 - iget);
}
```

*: Of course, such a function would also need to be declared in the `RingBuffer` class, which has been omitted here for brevity.

Using Templates for Policies

To pick up the example of the chapter on exceptions, here is another possibility shown how a client may determine the error handling policy of class `Mine`, using a template-based policy.

The `ErrorPolicy` private base class encapsulates the decision how to handle the error:

```
template<class ErrorPolicy>
class Mine : private ErrorPolicy {
    using ErrorPolicy::error_X;
public:
    void do_it() {
        …
        if (…) error_X(12, 15);
        …
    }
};
```

Using Templates for Policies (2)

Following are two examples for concrete policies:

```
class ThrowingPolicy {
    void error_X(int a, int b) { throw ErrorX(a, b); }
};

class NonThrowingPolicy {
    void error_X(int, int) { /*empty*/ }
};
```

Throwing and non-throwing versions of class `Mine`:

```
typedef Mine<ThrowingPolicy> ThrowingMine;
typedef Mine<NonThrowingPolicy> NonThrowingMine;
```

The advantage here is that especially in the non-throwing variant the empty function will generate no code at all, while with virtual member functions there would be a jump into a subroutine that just returns immediately.

Template Meta-Programming

As an advance example to motivate the necessity of meta-programming, look at the argument of the `RingBuffer::put` member function:

```
template<typename T, std::size_t N>
class RingBuffer {
    …
    void put(const T &e);
    …
};
```

Using a constant reference here is a safe choice but not a good solution in every case – in fact, small basic types are better handed over by value.

> What we would need were a function that turns the (generic) template argument `T` into either a `const T&` (for the general case) or leave it at `T` (especially for small basic types).

Boost Call Traits

Such a compile time function already exists (in a utility library of the Boost Platform) and would be used as follows:*

```
template<typename T, std::size_t N>
class RingBuffer {
    …
    void put(typename boost::call_traits<T>::param_type e);
    …
};
```

*: Only in case you are curious for the implementation, the basic strategy is laid-out here:

```
// first handle the general case – by default use a constant reference:
template<typename T> struct call_traits { typedef const T& param_type; };
// then template specializations for small types: bool, char, short, ...
template<> struct call_traits<bool> { typedef T param_type; };
template<> struct call_traits<char> { typedef T param_type; };
template<> struct call_traits<short> { typedef T param_type; };
…
// then maybe partial specialization for pointer types:
template<typename T> struct call_traits<T*> { typedef T param_type; }
```

Standard Library Overview

Frequently useful library components are:

- The class `std::string`
- Classes for Stream Handling
- Container Classes from the STL
- Algorithms from the STL

Standard String Class

The `std::string` is designed to be used much like a built-in type.

The following program fragment prints the largest word from its input (in lexicographical sort order):

```
string word;
string maxword:
while (cin >> word) {
    if (maxword < word) maxword = word;
}
cout << maxword << endl;
```

As the class `std::string` is not covered in much detail here feel free to ask for more information based on the Infographic "Standard Strings" in the appendix.

Classes for Stream Handling

Support for handling data streams – mainly dealing with the aspects of abstraction and transparent buffering – falls into two big groups:

- `std::istream` and `std::ostream` hosting all the input and output operations while decoupling the actual data source or sink.
- `std::ifstream` and `std::ofstream` using a classical (external) file as data source or sink.
- `std::istringstream` and `std::ostringstream` using an `std::string` as data source or sink.

> As the above classes are not covered in much detail here
> feel free to ask for more information based on
> the Infographic "I/O-Stream Basics in the appendix.

A Poor Man's Approach to TDD

Consider the following fragment that request a name as input and sends a greet of which this name is a part:

```
void say_hello()
    string name;
    cin >> name;
    cout << "hello, " << name << endl;
}
```

The first change required is to use function arguments instead of global stream objects:*

```
void say_hello(istream& in, ostream& out)
    string name
    in >> name;
    out << "hello, " << name << endl;
}
```

*: Arguments of `say_hello` might actually be named `cin` and `cout` and therefore shadow the global objects, but this is not recommended as it could quickly lead to misunderstandings.

A Poor Man's Approach to TDD (2)

Now some input can be prepared by using the

- `std::istringstream prepared_input`

while the output is received in the

- `std::ostringstream received_output`

and compared to the expectation:*

```
int main() {
    istringstream prepared_input("Mary");
    ostringstream received_output;
    // ----------------------------------------
       say_hello(prepared_input, received output);
    // ----------------------------------------
    assert(received_output.str() == "hello, Mary\n");
    cout << "** TEST PASSED **" << endl;
}
```

*: Of course, a serious automated test should make sure that the answer is not always `hello, Mary\n`, no matter what the input was ... :-)

Container Classes from the STL

The STL is designed around *Containers* and *Algorithm* with an *Iterator Abstraction* as glue between the former.

The container classes provided fall into two big categories:

- Sequential Containers
- Associative Containers

> In C++ the term container class denotes a class which is capable to store object (instances) of some other class.

Sequential Containers

C++98 defined the following three:

- `std::vector` – contiguously stored objects (like in an native array) with providing random access and insertion/removal **at one end** with O(1) performance; existing elements are copied or moved to newly allocated memory, if necessary.
- `std::list` – non-contiguously stored objects with two links per element, providing insertion/removal with O(1) performance **anywhere** (even for other lists).
- `std::deque` mixture of the above (objects store in array-like chunks), combining random access and insertion/deletion at **both ends** (but not in the middle.

C++11 added the following:

- `std::array` – light-weight wrapper providing sequential container for native arrays.
- `std::forward_list` – non-contiguously stored objects with one link per element, providing less overhead for small payload per element.

Associative Containers

C++ 98 defined the following four (all contained elements stored in a balanced binary tree):

- `std::set` – "key"-only container without duplicates, providing look-up with O(log(N)) performance;
- `std::multiset` – as before but allowing duplicates.

- `std::map` – "key-value"-pair container without duplicates, providing look-up with O(log(N)) performance;
- `std::multimap` – as before but allowing duplicates.

C++11 added the following four (all contained elements stored in a hash-based bucket-chain structure):

- `std::unordered_set` – "key"-only container without duplicates, providing element-lookup with O(1) performance; performance for element look-up;
- `std::unordered_multiset` – as before but allowing duplicates.

- `std::unordered_map` – "key-value"-pair container without duplicates, providing element look-up with O(1) performance;
- `std::unordered_multimap` – as before but allowing duplicates.

Generic Container Iterators

All STL containers provide iterators, abstracting the way to pass over all contained elements.

A basic code fragment using iterators is sketched below:

```
vector<int> data;
…
for (vector<int>::iterator it = data.begin();
                           it != data.end(); ++it)
    … *it … // access element via dereferenced iterator
```

Using the new meaning of the keyword `auto` as defined by C++11 this could slightly abbreviated:*

```
for (auto it = data.begin(); it != data.end(); ++it) …
```

*: To forward iterate over all the content of a container, a C++11 range-based loop is even more compact:
`for (auto x : data) … x … // access element via placeholder x`

Iterating Over Maps

When iterating over a map, the iterator represents an `std::pair` of the key and the associated value. The following is a program that counts the number of occurrences of all the words in some text:*

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main() {
    map<string, int> words;
    string w;
    while (cin >> w)
        ++words[w];
    for (auto it = words.cbegin(); it != words.cend(); ++it)
        cout << it->first << ": " << it->second << endl;
}
```

*: In the first book about the C Programming Language by B.Kernighan and D.M.Ritchie this program served as demonstration for handling dynamic data structure and had a total of approximately 100 lines.

STL Algorithms

The STL specifies a number of algorithms for some typical processing requirements of elements in a container.

The following fragment could be used to sort the contents of a container:

```
vector<int> data;
…
sort(data.begin(), data.end());
```

It is a common property of all algorithms that a container is specified by a pair of iterators.

Output to Containers

Some algorithms allow to store their output in another container. Often a "cookbook"-style approach makes sense:

```
int arr[100];
… // fill in n elements
std::vector<int> data;
std::unique_copy(&arr[0], &arr[n], std::back_inserter(data));
```

Algorithms – A Look Inside

To really understand STL algorithms it makes sense to have a short look on the implementation of a typical example:*

```
template<typename InIt, typename OutIt, typename Pred>
OutIt filter(InIt from, InIt upto, OutIt dest, Pred pred) {
    while (from != upto) {
        if (pred(*from))
            *dest++ = *from;
        ++from;
    }
    return dest;
}
```

An algorithm like this is available as `std::copy_if` since C++11.*

*: In C++98 it seemed to be missing but what you can do with `std::copy_if` was already possible with `std::remove_copy_if` and the predicate negated, so `filter` from above might also be implemented as:

```
template<typename InIt, typename OutIt, typename Pred) {
OutIt filter(InIt from, InIt upto, OutIt dest, Pred pred) {
    return std::remove_copy_if(from, upto, dest, [pred](const auto& e) {return !pred(e);});
    //          so that you don't miss the important thing - it's HERE --------^
}
```

Specifying Predicates

There are many algorithms like `filter` requiring a predicate, e.g.:*

```
// count all elements in `arr` with positive values:
bool gt_zero(int x) { return (x > 0); }
…
auto n = count_if(&arr[0], &arr[n], gt_zero);
```

The basic technique is always the same:

```
// copy all non-zero elements from `arr` to `data`:
bool non_zero(int x) { return (x != 0); }
…
filter(&arr[0], &arr[n], std::back_inserter(data), non_zero);
```

---

*: While the above example code uses classic functions an alternative technique in widespread use would define the predicate as *Function Object* (or *Functor* in short):

```
// note object construction by appending `()` when the functor's type is used further below!
struct GtZero { bool operator()(int x) { return (x > 0); }};
struct NonZero { bool operator()(int x) { return (x != 0); }};
…
auto n = count_if(&arr[0], &arr[n], GtZero());    // or GtZero{} or NonZero{}  with C++11
filter(&arr[0], &arr[n], std::back_inserter(data), NonZero()); // uniform initialization
```

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*.

For the `filter` algorithm 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.

Lambda 101 – Argument Lists

The argument list and return value of a Lambda must always be compatible with the expectations of the caller.

- Typically algorithms with a predicate argument hand over a container element as argument and expect a boolean return value.
- Most notable exceptions are algorithms expecting, establishing or maintaining a sort order:
  - such hand over two container elements as argument and
  - expect `true` as return value if – for proper sort order – the first argument is to be placed before the second.

```
std::vector<double> data;
…
// sort doubles ascending (= default)
std::sort(data.begin(), data.end());
…
// sort doubles descending
std::sort(data.begin(), data.end(),
          [](double e1, double e2) { return (e2 < e1); });
```

While for more complex tasks a named function or functor is preferable for readability and maintainability, named lambdas are also worth a consideration as their point of definition can be close to their use:

```
…
// sort doubles by their absolute values
auto fabs_cmp = [](double e1, double e2) {
                    return (std::fabs(e1) < std::fabs(e2));
                };
…
std::sort(data.begin(), data.end(), fabs_cmp);
```

This is not limited to lambdas used only one time.*

*: If named lambdas are written for potential reuse it also makes sense to strive for a more generic solution. C++14 allows using `auto` as argument type:

```
// sort pairs by member `first` or `second`:
auto cmp_by_first = [](const auto& e1, const auto& e2) { return (e1.first < e2.first); };
auto cmp_by_second = [](const auto& e1, const auto& e2) { return (e1.second < e2.second); };
…
std::sort(data.begin(), data.end(), cmp_by_first);
std::stable_sort(data.begin(), data.end(), cmp_by_second);
```

To emphasize that important point once more: for any function (e.g. `filter`) expecting some other function as argument,

- the lambda handed over by the caller (e.g. *as argument to* `filter`)
- must be callable by the code inside (i.e. *the implementation of* `filter`).

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

```
void foo(std::vector<int> data, std::list<int> result, int max) {
    filter(data.begin(), data.end(), std::back_inserter(result),
           [](int e, int max) { return (e < max); });
};
```

[!] This code will not compile because the lambda doesn't match the expectation `filter` has about its fourth argument.

*: How the error manifests in a compiler diagnostic is a different issue: for a fully generic template the compiler will typically give a diagnostic for the location where the actual call occurs and also point to the code by which that particular instantiation was caused.

Lambda 101 – Capture List Example

In a capture list variables from the local context are named.

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

```
void foo(std::vector<int> data, std::list<int> result, int 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 path you will get an idea when Classic C++ Function Objects are covered.

Beware of the Pitfalls with Lambdas

There are two broad categories of pitfalls:

- Not being aware of the fact that algorithms have the freedom to make a copy of a lambda, also copying its local data.
- Initializing a reference in a long-living lambda with a short(er)-living object instance or basic variable.

As in both cases the program will not come to the expected result, inadvertently making a copy usually cause less harm.

> **[!]** Via dangling references unrelated memory locations may be accessed and even modified, sometimes causing no effect until long after the fact.

Classic C++ Function Objects

Before C++11 introduced lambdas *Function Objects* or *Functors*[*] were the way to go, if besides the arguments passed by the caller additional information had to be passed to a piece of callable code.

```
class IsBelow {
    int max;
public:
    IsBelow(int m) : max(m) {}
    bool operator()(int m) const { return (e < max); }
};
…
void foo(std::vector<int> data, std::list<int> result, int max) {
    filter(data.begin(), data.end(), std::back_inserter(result),
           IsBelow(max));
}
```

Type Generic Functors

Note that also Functors can be made type generic:

```
template <typename T>
class IsBelow {
    T max;
public:
    IsBelow(T m) : max(m) {}
    bool operator()(T m) const { return (e < max); }
};
```

This is rawly equivalent to `auto` arguments in lambdas, though there still needs to be a type argument:*

```
void foo(std::vector<int> data, std::list<int> result, int max) {
    filter(data.begin(), data.end(), std::back_inserter(result),
           IsBelow<int>(max));
}
```

---

*: This is expected to be alleviated in C++20. Until the following helper function may come in handy:
`template<typename T> IsBelow<T> gIsBelow(T v) { return IsBelow<T>(v); }`
What turns the call of `filter` into:
`filter(data.begin(), data.end(), std::back_inserter(result), gIsBelow(max));`

Local Context by Reference (Functors)

Both, Lambdas and Functors provide ways to access variables from the local context by reference.

What needs to happen behind the scene is obvious in a functor:

```
template<typename T>
class PrintEnumerated {
    std::ostream& os;
    int& num;
public:
    PrintEnumerated(std::ostream& o, int& n) : os(o), num(n) {}
    void operator()(const T& e) { os << n++ << '\t' << e << '\n'; }
};
…
void foo(std::vector<int> data, std::ostream output) {
int lnum = 1;
std::foreach(data.cbegin(), data.cend(),
             PrintEnumerated<int>(output, lnum));
}
```

... but much of the code (adding the necessary members and a constructor forwarding its arguments) is just boiler-plate.

Local Context by Reference (Lambdas)

A lambda makes the code more succinct and to the point:*

```
void foo(std::vector<int> data, std::ostream &output) {
    int lnum = 1:
    std::foreach(data.cbegin(), data.cend(),
                 [&output, &lnum](int e) {
                     output << ++lnum << '\t' << e << '\n';
                 });
}
```

*: The code could be made even more compact by abbreviating the capture list to a single &:
`[&](int e) { output << lnum++ << ":\t" << e; }`

Generalized Callables

Summarizing there as three forms of callable code:

- Classic C function pointers
- C++ Callable Objects* aka. Functors
- Lambdas introduced in C++11

All three are different types and hence can only vary if specified directly or indirectly through a template argument.

> The template `std::function` provides a common wrapper.

Argument and result types need to be to be fixed, though – *callables need not match exactly,* only within the limits of the standard conversions, (as are supplied for any direct function call too).

---

*: An object is *callable* if after any of its instances a pair of parentheses enclosing an (optionally empty) argument is acceptable. This is achieved by overloading `operator()`. **Note that in its definition the formal argument list is supplied inside an extra set of parenthesis.**

Use of Callables

There are two main applications:

- Replacing a fully general template argument for an internal variation of some algorithm via an argument.
  - Usually this trades in a loss of runtime performance for a smaller memory footprint.
  - It also allows the compiler to give better error messages when the call is malformed.
- When a variable (or a container) needs to refer to the actual code.
  - This could be call-backs in asynchronous (message based) designs, or
  - some loose coupling of encapsulated software components*

*: In the world of design patterns this is also known as *Publisher-Subscriber*.

First Code Example with `std::function`

This example shows how a predicate specified as fully generic type (though of course bounded by its use) ...

```
template<typename T, P>
std::size_t count_value_if(const std::vector<T> data, P pred) {
    std::size_t result = 0;
    for (auto e : data)
        if (pred(e)) ++result;
    return result;
}
```

... can be replaced with an `std::function`-type argument:

```
template<typename T>
std::size_t count_value_if(const std::vector<T> data,
                           std::function<bool(int)> pred) {
    // … same as before
}
```

The difference shows in the generated code but becomes even more obvious in the error messages in case the actual callable has an incompatible signature.

Second Code Example with `std::function`

The following example demonstrates a bare-bones implementation of a communication scheme in the spirit of the publisher-subscriber pattern:

```
struct slot_call : std::vector<std::function<void(const char*)> {
public:
    void operator()(const char *arg) const {
        for (const auto &fnc : *this) fnc(arg);
    }
};
```

The intended use is like this:

```
slot_call sc;                 // publisher object defined here
…
                              // subscribers register handlers
sc.push_back([](const char* s) { /* ... like here ... */ });
sc.push_back([](const char* s) { /* ... and there ... */ });
…
while (auto cp = get_events()) // then publisher listens to ...
    sc(cp);                    // ... and distributes events
```

`std::function` Performance Considerations

When `std::function` is involved all **actual calls** will typically take place via pointer indirection.

> A call via `std::function` is comparable to a `virtual` member function.

If alternative implementations are possible, the rule of thumb is this:

- An `std::function`-type, i.e. a callable *bound in its definition* to a fixed signature, tends to
  - generate **less**
  - but **slower** code.
- A generic templated type, only bounded *by its use as callable*, tends to
  - generate **more**
  - but **faster** code.

`std::bind` vs. Lambdas

Lambdas are also useful to adapt more generic functions to a specific use.

This may be any or a combination of

- binding one or more arguments to fixed values
- binding arguments in a different order.

The following examples assumes a

- `drawing` composed of
- `point`s and taking a
- `pen` to create a visual representation:

```
struct point { int x, y; };
struct drawing : private std::vector<point> {
    using pen_t = std::function<void(double x, double y)>;
    void draw(pen_t pen) {
        for (const auto pt : *this) pen(pt.x, pt.y);
    }
};
```

Code Example with `std::bind`

Further assuming there is the following function

```
void my_test_pen(std::ostream& os, int yc, int xc) {
    os << "x= " xc << ", y=" << yc << '\n';
}
```

which is not exactly but close to the expected `pen`.

```
drawing d = {{0,1}, {3,7}, {4,6}, {1,1}, {0,1}};
…
using namespace std::placeholders;
d.draw(std::bind(my_test_pen, std::ref(std::cout), _2, _1));
```

Alternatively a lambda may be used:*

```
d.draw([](int x, int y) { my_test_pen(std::cout, y, x); });
```

---

*: Note that `std::cout` is a global object instance and therefore directly available in a lambda body. If the stream to use would come from a local variable or function parameter `std::ostream os` and the lambda arguments were renamed to `_1` and `_2`, the comparison would come even closer:
`d.draw(std::bind(my_test_pen, std::ref(os), _2, _1));` vs.
`d.draw([&os](int _1, int _2) { my_test_pen(os, _2, _1); });`

More Pitfalls with Callables

A general pitfall when using callables returned from `std::bind` or created with lambdas is that C++ has no garbage collection.

> **[!]** Especially if the callables are handed over back and forth, the code must be carefully designed and should be reviewed with life-time issues in mind!

This was (obviously?) not the case for the following two fragments.*

Houston, we have a problem ...

```
void g(std::ostream&, int);
…
std::function<void(int)> f() {
    using namespace std;
    using placeholders::_1;
    ostringstream os;
    return bind(g, ref(os), _1);
}
```

... similar over here:

```
void g(std::ostream&, int);
…
std::function<void(int)> f() {
    std::ostringstream os;
    return [&os](int x) {
                g(os, x);
           };
}
```

*: So that the problem manifests itself there must of course be a call like this: `f()(42);`

Library Extensions since C++11

This section summarizes some important extensions of the C++ standard library introduced with C++11, C++14, and C++17.

- Durations, Time Points, and Clocks
- Pseudo Random Numbers (and Distributions)
- Regular Expressions
- Ad-hoc Data Structures: `std::tuple`
- Run-Time Checked Generic Pointer: `std::any`
- Single-Element Containers: `std::optional`
- Type-Tracking Unions: `std::variant`

Durations, Time Points, and Clocks

C++11 introduced a set of new classes in the sub-namespace `std::chrono` to abstract

- Time Points and
- Durations.

As for some similar libraries operators are overloaded for meaningful operations only.

E.g. you may:

- add or subtract a durations to/from a duration
- multiply or divide a duration with/by a (dimensionless) numbers
- divide a duration by a duration (yielding a dimensionless number)
- add or subtract a duration to/from a time point (yielding a time point)
- subtract a time point from a time point (yielding a duration)

But you may **not**:

- mix durations and time points in any other operation but subtraction
- mix time points and dimensionless numbers in **any** operation

Duration Example

While a generic template allows to specify a *Duration Type* in detail, most often some predefined type (based on the generic template) is used.

The following example may help to understand the most basic use:

```
namespace sc = std::chrono;
…
auto dur = sc::hours(3)
         + sc::minutes(17)
         + sc::seconds(12);
```

Here the variable `dur`

- is determined to have type* `std::chrono::seconds`
- with the (initial) value `3*3600 + 17*60 + 12` (= 4632).

Via *Overloading Suffix Operators* C++14 simplified this further (see also http://en.cppreference.com/w/cpp/symbol_index/chrono_literals):

```
using namespace std::chrono_literals;
…
auto dur = 3600h +17min + 12s;
```

*: This reveals another aspect of the elegant design: Template Meta-Programming is used in the implementation so that a combination of different duration types always will chose a result type according to the operand with the finest-grained resolution (which is seconds here).

Clock Example

Time Points often involve using a *Clock* to get the current time.

C++11 requires the following three clocks as a minimum* (see also http://en.cppreference.com/w/cpp/chrono#Clocks):

- `std::chrono::system_clock`
- `std::chrono::monotonic_clock`
- `std::chrono::high_resolution_clock`

Measuring the execution time of some piece of code could basically be done as shown here:

```
namespace sc = std::chrono;
const auto start = sc::high_resolution_clock::now();
…
… // code to time
…
const auto end = sc::high_resolution_clock::now();
using show_duration_ms = sc::duration<double, std::ratio<1, 1000>>;
std::cout << show_duration_ms(end-start) << "ms" << std:::endl;
```

*: Implementations are free to extend this, e.g. standard conforming Boost.Chrono adds clocks running only when the current process uses the CPU, allowing to measure CPU load instead of real-time.

Pseudo Random Numbers

For compatibility with C the following simple approach still works:

- generate random numbers with `rand()`,
- eventually after seeding the generator with `srand()`

The new design in C++ separated parating (random) generators from distributions, so throwing "a fair dice" requires this:*

```
int dice() {
    static std::default_random_engine random_roll{};
    static std::uniform_int_distribution<int> one_to_six{1, 6};
    return one_to_six(random_roll);
}
```

*: While having some more lines of code it avoids the well-known deficiencies of classic much simpler approach `std::rand()%6 + 1`: the sometines very short cycle until repetition (with 100% predictability then) and the tendency to prefer the smaller values because of truncation with the modulo operator.

Regular Expressions

The following fragment shows how to break down an `std::string` assumed to hold a currency content formatted

- with full Euros into "3-digit groups" separated by dots
- and Eurocents optional, separated by a comma if present:*

```
bool euro_parser(const std::string& str, double& ec) {
    std::regex re{R"(([1-9]\d{0,2}(?:\.\d{3})*),(\d{2}))"};
    std::smatch m;
    if (!std::regex_match(str, m, re))
        return false;
    const std::string es{m[1].str()};
    ec = std::stod(std::regex_replace(es, std::regex{"\\."}, ""))
       + std::stod(m[2].str())/100.0;
    return true;
}
```

The topic is only covered further when asked for.
Then some more overview can follow based on the
Infographic "C++11 Regular Expression API" in the appendix.

*: The regular expression has a minor bug which shows for amounts of under 1€.

Ad-hoc Data Structures: `std::tuple`

The data type `std::tuple` supports to combine usually separate pieces of information into a single unit without requiring to define a specific `struct` or `class`.

A tuple can be

- constructed,
- filled,
- accessed, and
- unpacked

in several ways, furthermore ...

It is also possible to
process an `std::tuple` element by element
at compile time

... using the techniques of *Template Meta-Programming*.

Except for the Meta-Programming aspect an `std::tuple` is an extension of `std::pair` (from C++98) and there are also conversions between both.

For more information see:
http://en.cppreference.com/w/cpp/tuple)

`std::tuple` Typical Implementation

Behind the scenes `std::tuple` is implemented using compile-time recursion, but this is invisible for the users.

Generally speaking:
There is no reason why an `std::tuple` should have a

- larger memory footprint or
- worse runtime performance

compared to a `struct` holding the same data members.

In practice this means:

- there **may or may not** be substantial differences if a program replaces small "one-time use" data structures by `std::tuple`-s;*
- the memory layout of **may or may not** be binary-compatible with an equivalent `struct`.

*: The C++11 standard does not specify any guarantee so this is rather a "*Quality of Implementation*" (= QOI) Issue. The free Compiler Explorer in the Internet may be useful to find out more ...

`std::tuple` Example

Some examples for using an `std::tuple` are following. They are based on the `euro_parser` function shown before and combine two informations in the value returned:

- a `bool` indicating whether a conversion was possible
- a `double` holding the converted amount

While the modifications to the original `euro_parser` are not that substantial, it is shown once more in completion here:*

```
auto euro_parser(const std::string& str) {
    using namespace std;
    regex re{R"((0|[1-9]\d{0,2}(?:\.\d{3})*),(\d{2}))"};
    smatch m;
    if (!regex_match(str, m, re))
        return make_tuple(false, 0.0);
    const auto es{m[1].str()};
    const auto ec = stod(regex_replace(es, regex{"\\."}, ""))
                  + stod(m[2].str())/100.0;
    return make_tuple(true, ec);
}
```

*: Note that also the problem with the original regular expression has been fixed here.

`std::tuple` Example (2)

Also the new C++11 suffix return type syntax may be tried if the long-winded return type specification preceding the function name and argument list is found to be distracting:

```
auto euro_parser(const std::string& str)
    -> std::tuple<bool, double> {
    … // as before
}
```

Since C++14 the type value will be deduced from the return statement, so the function return type can be omitted if the return statements are modified to use

- either the tuple constructor explicitly ...
  - `return tuple<bool, double>{false, 0.0};`
  - `return tuple<bool, double>{true, ec};`
- ... or the factory function for tuples:
  - `return make_tuple{false, 0.0};`
  - `return make_tuple{true, ec};`

`std::tuple` Example (3)

On the receiving end the tuple must be stored and unpacked:

```
std::string input;
…
const auto result = euro_parser(input);
if (std::get<0>(result)) // branch based on success
    … std::get<1>(result) … // access amount
else
    …                       // conversion failed
```

Since C++14 members of a tuple can also be accessed by their type name (if unique inside the tuple);

```
if (std::get<bool>(result))
    … std::get<double>(result) …
else
    …
```

`std::tuple` Example (4)

Alternatively unpacking can be done with `std::tie`:

```
bool success;
double euro;
std::tie(success, euro) = euro_parser(input);
if (success) {
    … // conversion ok, amount found in `euro`
else
    … // (invalid input format)
```

If not all members of a tuple are of interest, a tie can also be done only partially, e.g. if only a validity check of the input format is desired …

```
bool success;
std::tie(success, std::ignore) = euro_parser(input);
```

… or if a zero-amount of is acceptable to indicate a failed conversion:

```
double euro;
std::tie(std::ignore, euro) = euro_parser(input);
if (euro != 0.0) … // conversion ok, amount found in `euro`
```

Single-Element Containers: `std::optional`

This class, introduced in C++17,* offers a simple standard way to add a "validity flag" to any kind of value.

A typical use case for `std::optional` is to indicate the "*not valid*" or "*not (yet) initialized*" state for a type that uses all bit patterns of its binary representation so that there is no special value left to serve as sentinel.

E.g. a function to convert a character string into a numeric value could use an `std::optional<int>` as return value to indicate a failed conversion.

For more information see:
http://en.cppreference.com/w/cpp/utility/optional

*: Prior to this it was introduced in the C++ in 2015 with the Library Fundamentals TS as `std::experimental::optional` and before was available since long as Boost.Optional.

`std::optional` Typical Implementation

An object of type `std::optional<` *T* `>` locates its payload (*T*) in proximity with a validity flag

- on the stack,
- at a static address,
- or on the heap

just as it would be the case for any ordinary object or base type variable.

`std::optional` Example

In the `euro_parser` it could have been used as return value too:

```
auto euro_parser(const std::string& str)
    -> std::optional<double> {
    std::regex re{R("([1-9]\d{0,2}(?:\.\d{3})*),(\d{2}))"};
    std::smatch m;
    if (!std::regex_match(str, m, re))
        return {};
    const std::string es{m[1].str()};
    return std::stod(std::regex_replace(es, std::regex{"\\."}, ""))
         + std::stod(m[2].str())/100.0;
}
```

`std::optional` Example (2)

There is a default `bool`-conversion to check validity while accessing the payload will be done with the `get` member function:

```
…
if (auto result = euro_parser(input)) {
    …
    … result.get() … // access converted amount
    …
}
else {
    …                // conversion failed
}
```

More Use Cases for `std::optional`

Besides marking the validity of a returned value there are some more typical use cases:

- clearly indicate the difference between
  - variables not yet initialized and
  - variables to which a value has been assigned
- for function arguments support true "out"-parameters

To illustrate the latter, consider this:*

```
void f(const T1& in_only, T2& in_out, std::optional<T3>& out_only) {
    … in_only …; // `const` reference enforces read-only access
    … ++in_out …; // non-`const` reference allows modification
    out_only = T3{ … , … }; // assign (new) value as necessary
}
```

*: At first glance this example may seem over-engineered – shouldn't an ordinary (non-`const`) reference `T3&` suffice as `out_only` argument? Most often this may be true or make no difference, but using an `std::optional<T3>&` may still have a real advantage if there is no reasonable way to construct the receiving argument variable before. And even if there is, it may be inefficient to first construct something only to have it overwritten soon after.

Generic Pointers with Runtime-Checking): `std::any`

This class, introduced in C++17,[*]) offers an efficient replacement for generic, **un-typed** pointers (`void *`).

This is especially true if

- the type of such pointers needs to be tracked anyway, to apply
- a type-cast for accessing the pointed to object beyond the bit-level.

A typical use case for `std::any` is in layered software architecture where some "middle" layer strives to reduce dependencies.

E.g. generic communication service may want to be "agnostic" with respect to a set of types well-known at the more special data collecting, processing, and presentation layers, for which it provides the connection.

For more information see:
http://en.cppreference.com/w/cpp/utility/any

[*]: Prior to this it was introduced in the C++ in 2015 with the Library Fundamentals TS as `std::experimental::any` and before was available since long as Boost.Any.

`std::any` Typical Implementation

As the maximum payload cannot be known at compile time, `std::any` needs to use heap allocation – al least as a fall back for the general case.*

The **space overhead** beyond the payload and the access pointer is

- a type discriminator, and
- some heap management data.

(The latter is often shared between a number of heap objects and so cannot be easily attributed to a single object.)

The **runtime overhead** is that of an pointer plus some

- validity checking for retrieval, and
- some resizing heap operations for content type changes.

(Validity checks for retrieval include returning an error indication if the actual and the expected content type differ.)

---

*: The programming interface does not exclude some kind of SOO (= Small Object Optimization) being used behind the scenes, similar to what is often found in `std::string` implementations on machines with large (64 bit) word size.

`std::any` Example

```
std::any v; // `v` may later hold any (moveable or copyable) type
```

Assuming the above and everything required to be done when storing new content is hidden in an overloaded assignment operator,* then

```
v = 42;
```

... assigns an `int` ...

```
v = std::string("hello");
```

... as easily as an `std::string`.

While the default c'tor (without arguments) puts the `std::any` in an "empty" state, a value may also be assigned on construction, so ...

```
std::any v2{42};
```

... initializes a value of type `int` ...

```
std::any v3{"hello"};
```

... or type `const char *`.

---

*: The final specification according to C++17 has slightly modified and enhanced the programming interface originating from Boost.Any mainly to allow optimizations when *Moving* is cheaper than *Copying*.

`std::any` Example (2)

Furthermore the `std::any` member function

- `reset` puts it back to the "not initialised/assigned" state, and
- `has_value` may be used to test for that particular state.

The more interesting part is how content is accessed.

> On each retrieval the expected content type needs to be specified.

There are two forms which differ in their failure indication, i.e. if the current content type is different from the expressed expectation.

`std::any` Example (3)

The first technique indicates failure by throwing an exception of type `std::bad_any_cast` and applies when

- at a given point in the flow of program execution there is only one valid content type, and
- everything else would be a programming error requires fixing (or maybe some kind of soft recovery strategy).

```
try { // expecting `v` contains a `double` number
    …
    … std::any_cast<double>(v) … // retrieve and process
    …
}
catch (std::bad_any_cast& ex) { // `v` contains anything else
    …
}
```

Of course, if nothing can (or needs) to be done at that point the exception should simply be let propagate down the call stack.

`std::any` Example (4)

The second technique explicitly tests for in a series* for content types:

> Note the `any_cast` here is applied to `&v` (address of `v`) instead of plain `v`.

```
…
if (!v.has_value()) {
    … // handle the "empty" case
}
else if (auto p = std::any_cast<int>(&v)) {
    … *p … // refers to contained `int` value
}
else if (auto p = std::any_cast<double>(&v)) {
    … *p … // refers to contained `double` value
}
else if (auto p = std::any_cast<MyClass>(&v)) {
    … *p …(or)… p-> … // refers to contained `MyClass` instance
}
…
```

A final else may provide a default (without retrieval) or indicated failure.

*: The chain may be run-time optimized by ordering the tests according to probability.

Type-Tracking Unions: `std::variant`

This class, introduced in C++17,* offers an efficient replacement for overlay-types like C-style `union`-s.

This is especially true if *the currently held content-type needs to be tracked by the program logic anyway,* to select the correct member before the content can be accessed (other than just at the bit-level).*

> A typical use case for `std::variant` are containers holding a bounded set of element types as content.

The advantage over `std::any` is that the "type-aware" parts of the system can be checked for completeness at compile-time, helping to diagnose problems early if the list of types to support grows.

> (i) For more information see:
> http://en.cppreference.com/w/cpp/utility/variant

*: Prior to this it was introduced in the C++ in 2015 with the Library Fundamentals TS as `std::experimental::variant` and before was available since long as Boost.Variant.

`std::variant` Typical Implementation

As the maximum payload is known at compile time, `std::variant` can use stack or fixed-address allocation.

The **space overhead** beyond the payload is

- a type discriminator, **plus**
- the size difference between the actual and the largest payload.

(So, if there is a vast size difference of payloads and the largest one is only rarely used, `std::any` might be the better choice.)

The **runtime overhead** is that of

- some validity checking for retrieval, and
- a d'tor call for the current type when assigning a different one.*

(It is arguable whether the latter should be considered as an overhead because every instance of a class needs to be destructed at sometime.)

---

*: If such an assignment can fail the `std::variant` will not provide Strong Exception Safety but instead leave the instance in a "value-less state", which can be tested for with the `valueless_by_exception()` member function.

`std::variant` Example

The set of types that can be stored in a variant is called "bounded" because it is fixed at compile-time …

```
std::variant<bool, float, std::string> v; // `v` may hold a `bool`
                                          //     OR a `float`
                                          //     OR an `std::string`
```

Assuming the above and everything required to be done when storing new content is hidden in an overloaded assignment operator,* then

```
v = 4.0f;
```

(assigns a `float`)

```
v = std::string("hello");
```

(assigns to `std::string`)

Due to automatic type conversions also the following works:

```
v = 4;
```

```
v = "hello";
```

`std::variant` Example (2)

But there are also c'tors accepting a single argument of each of the other types, so ...

```
decltype(v) v2{4.2};
```

... initializes it as `float` ...

```
decltype(v) v3{"hello"};
```

... or type `std::string`.

An "un-initialized state" is possible, but **not** with the default c'tor (without arguments) as this initializes an `std::variant` according to its first type.*

```
std::variant <void*, unsigned long, MyClass> x; // <== nullptr
std::variant <MyClass, void*, unsigned long> y; // <== MyClass{}
std::variant <unsigned long, MyClass, void*> z; // <== 0uL
```

[!] The un-initialized state is assumed if an assignment throws after the old content has been already cleared-out and a failure occurs before the new content is completely written.

*: Built-in types will get the same initialization as for globals.

`std::variant` Example (3)

On each retrieval the expected content type needs to be specified.

There are three forms which differ in their failure indication, i.e. if the current content type is different from the expressed expectation.

The first technique indicates failure by throwing an exception of type `std::bad_variant_cast`.*

```
try { // expecting `v` contains a `double` number
    …
    … std::get<float>(v) … // retrieve and process
    …
}
catch (std::bad_variant_access& ex) { // `v` contains anything else
    …
}
```

*: As for `std::any` this technique is usually preferable if the reason is a program bug which finally needs some fixing in the source code, re-compilation etc. The **`try – catch` may exist** and attempt to "resolve the problem" within the range of options to completely it, trying silent continuation with of without logging and some "self-healing", **or it may not exist** letting the process crash with a core file, and maybe restart it immediately or let an embedded device carry out a warm-boot …

`std::variant` Example (4)

The second technique explicitly tests for in a series* for content types:

> Note `get` here is replace with `get_if` **and**
> is applied to **&**v (address of v) instead of plain v.

```
…
if (auto p = std::get_if<bool>(&v)) {
    … *p … // refers to contained `int` value
}
else if (auto p = std::get_if<float>(&v)) {
    … *p … // refers to contained `double` value
}
else if (auto p = std::get_if<std::string>(&v)) {
    … *p …(or)… p-> … // refers to contained `std::string` instance
}
…
```

> **[!]** The problem here is that – if a new alternative is added to the variant – it gets silently ignored **or** an inappropriate default processing is chosen.

The third technique is helpful to turn "missing type cases" after a new alternative has been added into compile time errors.

This is done with a static variant of the Visitor Design Pattern. The first step is to define a Functor-like class overloading `operator()` with argument types matching the `std::variant` alternatives:*

```
struct my_variant_printer {
    void operator()(bool arg) const {
        std::cout << (arg ? 'T' : 'F');
    }
    void operator()(float arg) const {
        std::cout << arg;
    }
    void operator()(const std::string& arg) const {
        std::cout << '"' << arg << '"';
    }
};
```

*: The approach shown above was chosen to be backward compatible with Boost.Variant with only minor changes. With the extended facilities for Template Meta Programming in C++17 there are more options which may or may not be considered "even more elegant" – see here: http://en.cppreference.com/w/cpp/utility/variant/visit#Example

`std::variant` Example (6)

The Functor-like class then is

- handed over to the helper `std::visit` as first argument,
- with the variant instance to process as second argument*

```
std::visit(my_variant_printer{}, v);
```

A compile-time error will occur if the variant `v` holds alternatives which none of the overloaded `operator()` in `my_variant_printer` can handle.

*: With Boost.Variant **(1)** `boost::apply_visitor` is used instead of `std::visit`; **(2)** the Functor-like class has to be publicly derived from `boost::static_visitor<>`; **(3)** if some value is to be returned from the overloaded `operator()`s the type must be specified as template argument for `boost::static_visitor`.

```
// visitor-based processing of a variant (with compile-error for "forgotten" alternatives)
boost::variant<bool, …, …, MyClass> my_variant;
…
struct my_processing : boost::static_visitor<int>{
    int operator()(bool arg) const { …; …; return …; }
    …                                  // ^^^^^^^^--vvvvvvvv----- some `int`
    int operator()(const MyClass& arg) const { …; …; return …; }
};
…
int my_result = boost::apply_visitor(my_processing, my_variant);
```

C++11: Concurrency Basics

With C++11 support for multi-threading was introduced.*

- Parallelizing Independent Tasks
- Synchronisation with Mutexes
- One-Time Execution
- Messaging with Condition Variables
- Atomic Operations
- Direct Use of Threads
- Native Threading Model Handles
- Concurrency Recommendations

This optional part of the presentation was written with the intent to give an overview of the provided features only, it is by far not exhaustive!

*: At first glance concurrency features may appear "as just some more library classes and functions". But beyond the hood, and especially in the area of allowed optimisations and to provide Cache Coherence on modern multi-core CPUs, concurrency is closely intertwined with code generation issues.

Parallelizing Independent Tasks

For complex tasks which can be split into independent parts, concurrency **and scalability to multiple CPU cores** can be easily achieved by following a simple recipe:

1. Separate the task into a number of different functions (or calls of the same function with different arguments).
2. Run each such function by handing it over to `std::async`, storing the `future` that is returned (easiest in an `auto`-typed variable).
3. Fetch (and combine) the results by calling the member function `get` for each `future` instance.

That way all the functions may run concurrently and the last step synchronizes by waiting for completion.

*: Technically the return value of `std::async` is an `std::future` but as this is a template and the type is usually somewhat different to spell out, most usages of `std::async` store the result in an `auto`-typed variable.

Foundation: Futures and Promises

The foundation on which parallelizing tasks is build are Futures and Promises.

These need not be fully understood to apply the API (exemplified on the next pages), but may help to understand the basic machinery:

- A future is the concrete handle which a client can use to fetch the result, presumably made available by a different thread of execution.
- A promise is a helper class which may be used in a separate thread to make a result available for a client.

> (i) For more information on promises and futures see:
> http://en.cppreference.com/w/cpp/thread/future
> http://en.cppreference.com/w/cpp/thread/promise
> http://en.cppreference.com/w/cpp/thread/packaged_task

Parallelizing Example

The following example calculates the sum of the first `N` elements in `data` by splitting the work of `std::accumulate`, into two separate function calls, that may run concurrently:*

```
// calculate sum of first N values in data
//
long long sum(const int data[], std::size_t N) {
    auto lower_part_sum = std::async(
        [=]{ return std::accumulate(&data[0], &data[N/2], 0LL); }
    );
    auto upper_part_sum = std::async(
        [=]{ return std::accumulate(&data[N/2], &data[N], 0LL); }
    );
    return lower_part_sum.get()
         + upper_part_sum.get();
}
```

---

*: Note the use of lambdas above – the actual call to `std::accumulate` will only happen when the lambda gets executed! A similar effect can be achieved as follows:

```
// preparing the callable for std::async with std::bind:
… std::async(std::bind(std::accumulate, &data[0], &data[N/2], 0LL)) …
… std::async(std::bind(std::accumulate, &data[N/2], &data[N], 0LL)) …
```

Default Launch Policy

The default behavior is that the systems decides on its own whether an asynchronously started task is run concurrently.*

Using `std::async` **without an explicit launch policy**

- **is just a hint** that it is acceptable to run a callable unit of code concurrently,
- as long as it has finished (and possibly returned a result) latest when the `get`-call returns, which has bee invoked on the `std::future`.

> **[!]** Be sure **not** to use the default launch policy but specify concurrent execution explicitly (see next page) whenever it is essential that two callable run concurrently.

*: It is a well known effect that too many parallel threads of execution may rather degrade performance. Especially if threads are CPU bound, it makes little sense to have more threads as cores. Therefore the standard gives considerable freedom to the implementation, which might implement concurrency with `std::async` with a thread pool and turn to synchronous execution or lazy evaluation at a certain threshold.

Explicit Launch Policies

There is a second version of `std::async` which has a first argument to specify the launch strategy.

The standard defines two values:

- `std::launch::async`
  if set it **enforces** the callable will run on its own thread;*
- `std::launch::deferred` if set the callable will **not** run until `get` is invoked.

*: Generally, without specifying a launch policy, the runtime system decides when some callable started with `std::async` will effectively run – it may or may not be immediately. **A deferred thread start may cause deadlocks if the thread is part of a group of threads that need to synchronize with each other** (e.g. via Condition Variables).

Catching Exceptions

If the callable started via `std::async` throws an exception, it will appear as if it were thrown from the call to get.

Hence, if the asynchronously run task may throw, fetching the result should be done in a try block:

```
auto task1 = std::async( … ); // whatever-is-to-do (and may throw)
auto task2 = std::async( … ); // whatelse-is-to-do (and may throw)
…
try { … task1.get() … }
} catch ( … ) { // what may be thrown from whatever-is-to-do
   … // handle the case that whatever-is-to-do threw
}
try { … task2.get() … }
} catch ( … ) { // what may be thrown from whatelse-is-to-do
   … // handle the case that whatelse-is-to-do threw
}
```

Communication between Independent Tasks

First of all: If the need arises to communicate between independent tasks, this should be taken **as a strong warning** that such tasks are actually not independent.

[!] If parallel tasks are not independent, further needs follow quickly with respect to synchronize access to shared data ... **with all the further intricacies following from this**.*

Nevertheless there is one common case that requires a simple form of communication between otherwise independent tasks.

- If there are several tasks working towards a common goal
- of which one fails, making the goal unattainable,
- the others should not waste CPU-time needlessly.

*: In other words: Pandora's proverbial can of worms opens quickly and widely ...

Communicate Failure between Concurrent Tasks

A basic design that communicates failure between partners working towards a common goal is outlined in the following example.*

The workers could look about so ...

```
void foo(std::atomic_bool &die,
         … ) { // more args
    for ( … ) {
        if (die) return;
        …
        … // some complex
        … // algorithm
        …
        // may fail here
        if ( … ) {
            die = true;
            return;
        }
    }
}
```

... being run by that code:

```
std::atomic_bool die{false};
auto task1 = std::async(
    [&die, … ]{ foo(die, … ); }
);
auto task2 = std::async(
    [&die, … ]{ foo(die, … ); }
);
…
… task1.get() …
… task2.get() …
…
if (killed) {
    // goal not reached
    …
}
```

*: To keep this code simple it just returns in case of problems, though it requires only a few changes if problems should be communicated to the caller via exceptions.

Synchronisation with Mutexes

The word *Mutex* abbreviates *Mutual Exclusion* and describe the basic purpose of the feature.

- Allow only one thread to enter a Critical Section, typically non-atomically executed sequence of statements
- which temporarily invalidate an Class Invariant, or
- in other ways accesses a resource not designed for shared use.

In general, mutexes have at least two operations* for

- **lock**-ing and
- **unlock**-ing,

but frequently provide additional features to make their practical use more convenient or less error prone.

*: Though the operations may not be spelled exactly *lock* and *unlock* ... (especially as mutexes are somewhat related to [Semaphores], which originally named their lock (-like) acquire operation *P* and their unlock (-like) release operation *V*.

Mutex Example (1)

The following example calculates one table from another one:*

```
template<typename In, typename Out, typename Transformation>
void worker(const In data[], std::size_t data_size,
            Out result[], std::size_t &total_progress,
            Transformation func) {
    static std::mutex critical_section;
    while (total_progress < data_size) {
        critical_section.lock();
        constexpr auto chunks = std::size_t{100};
        const auto beg = total_progress;
        const auto end = ((data_size - total_progress) > chunks)
                 ? total_progress += chunks
                 : total_progress = data_size;
        critical_section.unlock();
        std::transform(&data[beg], &data[end], &result[beg], func);
    }
}
```

*: The work is shared by any number of `worker` task run concurrently, each fetches and transforms a fixed number of values. This can be advantageous to splitting the work by calculating fixed-size regions of the table in advance, if the transformation function has a largely varying runtime depending on the argument value.

Mutex Example (2)

Assuming the transformation is to calculate square roots and there are two arrays of size `N`, say

- `data` (filled with values to transform), and
- `sqrts` to store the results

workers may be created (to be handed over to `std::async`) as follows:*

```
std::size_t processed_count = 0;
auto worker_task =
    [&]() { worker(data, N, sqrts, processed_count,
                   [](double e) { return std::sqrt(e); });
    };
```

*: Given the above, a particular nifty way to create and run workers were:

```
// assuming NCORES holds the number of cores to use by workers:
std::array<std::future<void>, NCORES> workers;
for (auto &w : workers)
    w = std::async(worker_task);
for (auto &w : workers)
    try { w.get(); } catch (...) {}
```

Mutexes and RAII

As the mutex operations **lock** and **unlock** need to come correctly paired, they make a good candidate to apply a technique called RAII.*

It works by creating a wrapper class, executing

- the acquiring operation (or *lock*-ing in this case) in its constructor, and
- the releasing operation (or *unlock*-ing) in its destructor.

Such helper classes are available in C++11 `std::lock_guard`.

The big advantage is that unlocking the mutex is guaranteed for code blocks defining a RAII-style (guard) object locally, no matter whether control flow reached its end, or by `break`, `return`, or some exception.

*: This TLA is an abbreviation Bjarne Stroustrup once coined for *Resource Acquisition is Initialisation*. In a recent interview Stroustrup revealed that he is not particularly happy with the term he once choose. But to change it would require to travel back in a time machine and suggest something more appropriate to him, as today the term RAII is in much too widespread use to be replaced by something else.

Mutex Variants

Mutexes in C++11 come in a number of flavours, which controlling their behaviour with respect to the following details:

- Whether or not some thread that already locked a mutex may lock it once more (and needs to release it as often).
- Whether or not a thread waits if it finds a mutex locked by some other thread, and in the latter case until *when* (clock-based time point) or *how long* (duration) it waits.

For all mutex variants there are also variants of RAII-style lock guards.

For more information see:
http://en.cppreference.com/w/cpp/thread/recursive_mutex
http://en.cppreference.com/w/cpp/thread/timed_mutex
http://en.cppreference.com/w/cpp/thread/recursive_timed_mutex and
http://en.cppreference.com/w/cpp/thread/unique_lock

C++14: Upgradable Locks

C++14 added the class `std::shared_lock`, supporting a frequent necessity:*

> Multiple Reader/Single Write Locking Schemes

- Any number of (reader) threads may successfully `shared_lock` that kind of mutex ...
- ... but only one single (writer) thread is allowed to actually `lock` it (unshared).

C++14 also provides a RAII-style wrapper for shared locking.

> (i) For more information see:
> http://en.cppreference.com/w/cpp/thread/shared_timed_mutex
> and http://en.cppreference.com/w/cpp/thread/shared_lock

*: Note that the terms "reader" and "writer" indicates the typical use of that kind of mutex, assuming that it is sufficient for readers to obtain the lock shared, as it guarantees the invariants hold but no modifications are made, while writers will need to temporarily break invariants.

Defeating Deadlocks Caused by Mutex-Locking

As a potentially blocking mechanism mutexes are famous for creating deadlocks, i.e *in the situation where two resources* A *and* B *are required,* one thread acquires these in the order *first A, then B* and *another thread acquires these in the order* first B, then A*.*

The obvious counter measure is to acquire locks always in the same order, as achievable with `std::lock` and `std::try_lock`.

For more information see:
without creating the potential for dead-locks) see:
http://en.cppreference.com/w/cpp/thread/lock and
http://en.cppreference.com/w/cpp/thread/try_lock

*: In practice, the potential for deadlocks is often not as obvious as in this example but much more intricate and close to impossible to spot, even in scrutinising code reviews. So, if (accidental) deadlocks cannot be avoided, sometimes a "self-healing" strategy is applied that works follows:
If more than one lock needs to be acquired, acquire at least all others with setting a time-out. If that hits, **release all locks** acquired so far, **delay** for some small amount of time (usually determined in a window with some slight randomness), then **try again** (and maybe in a different order).

One-Time Execution

For a particular scenario that would otherwise require the use of mutex-es to avoid a Race Condition, there is pre-built solution in C++11.

Executing a piece of code exactly once can be achieved in a cookbook-style as follows:

```
// in a scope reachable from all usage points:
std::once_flag this_code_once;
…
std::call_once(this_code_once, …some callable… ); // somewhere
…
std::call_once(this_code_once, … ); // maybe somewhere else
```

For any of the callables associated with the same instance of an `std::once_flag` via `std::call_once`, without further protection via mutexes it is guaranteed that **at most one** is executed **at most once**.

For more information see:
http://en.cppreference.com/w/cpp/thread/once_flag and
http://en.cppreference.com/w/cpp/thread/call_once

One-Time Execution Example

A typical use case for guaranteed one-time execution is some initialisation, that may be expensive and is therefore delayed until a function that depends on it is called the first time.

The following fragment avoids parallel initialisations of `table`:

```
… foo( … ) {
    static std::once_flag init;
    static std::array<int, 1000> table;
    std::call_once(init, [&table]) {
        … // precalculate table when foo runs for the first time
    });
    … //
}
```

[!] Note that the above code still has a problem with respect to the initialisation it seems to guarantee …*

*: The example code shown does **not** guarantee that from several concurrently executing threads all will see a **fully** initialised table – it only guarantees sure that the callable to pre-calculate the content will be executed exactly once and from exactly one thread.

Local `static` Initialisation

Since C++11 supports multi-threading in the core language, initialising local `static` variables is protected to be executed at most once:

```
… foo( … ) {
   static const int z = expensive_calculation();
   … //
}
```

As since C++11 compilers are required to wrap the necessary protection around the initialisation of static locals, also this is guaranteed to work:*

```
class Singleton {
    … //
public:
    static Singleton &getInstance() {
        static Singleton instance;
        return instance;
    }
};
```

*: Non-believers should consider to copy the above code, paste it to https://gcc.godbolt.org/ (or similar) after adding a member with a runtime-dependant initialisation, and view the assembler output …

Notifications with Condition Variables

A well-known abstraction in concurrent programming are combining mutexes with a signalling mechanism.

One main use of condition variables is to **avoid busy waiting** in producer-consumer designs,

- where consumer and producer run concurrently,
- exchanging data over some buffer data structure.

For more information see:
http://en.cppreference.com/w/cpp/thread/condition_variable

Condition Variable Example (1)

The following *RingBuffer* class can put condition variables to good use ...

```
template<typename T, std::size_t N>
class RingBuffer {
    std::array<T, N+1> buf;
    std::size_t p = 0, g = 0;
    bool empty() const { return p == g; }
    bool full() const { return (p+1) % buf.size() == g; }
public:
    void put(const T &val) {
        if (full())
            … // handle case no space is available
        buf[p++] = val; p %= buf.size();
    }
    void get(T &val) {
        if (empty())
            … // handle case no data is available
        val = buf[g++]; g %= buf.size();
    }
};
```

... exactly at the currently omitted points.

Condition Variable Example (2)

Obviously there are two conditions, that need special attention:

- The buffer may be full when `put` is called, or
- it may be empty, when `get` is called.

Therefore two condition variables* are added, furthermore a mutex to protect accessing the buffer:

```
class RingBuffer {
    … //
    … // as before
    … //
    std::condition_variable data_available;
    std::condition_variable space_available;
    std::mutex buffer_access;
public:
    … // see next page
};
```

*: As the buffer space cannot be full and empty at the same time, technically one condition variable would suffice, but for this introductory example using two different instances seems to be clearer.

Condition Variable Example (3)

There are two operations (of interest here), applicable to condition variables, **sending** and **waiting for** notifications:*

```
class RingBuffer
   … // see previous page
public:
    void put(const T &val) {
        std::unique_lock<std::mutex> lock(buffer_access);
        space_available.wait(lock, [this]{ return !full(); });
        buf[p++] = val; p %= buf.size();
        data_available.notify_one();
    }
    void get(T &val) {
        std::unique_lock<std::mutex> lock(buffer_access);
        data_available.wait(lock, [this]{ return !empty(); });
        val = buf[g++]; g %= buf.size();
        space_available.notify_one();
    }
};
```

*: Essential here is also the connection between the condition variables, the mutex protecting the `RingBuffer` invariants, and the conditions checked as part of waiting, which are detailed on the next page.

Waiting Anatomy

Waiting on a condition variable – as shown on the last page – with

```
// as part of put:
   std::unique_lock<std::mutex> lock(buffer_access);
   space_available.wait(lock, [this]{ return !full(); });


// as part of get:
   std::unique_lock<std::mutex> lock(buffer_access);
   data_available.wait(lock, [this]{ return !empty(); });
```

is equivalent to the following, **with the mutex being locked before**:

```
// as part of put:
   while (full()) { // !!full()
       buffer_access.unlock();
       // wait for notification
       buffer_access.lock();
   }
```

```
// as part of get:
   while (empty()) { // !!empty
       buffer_access.unlock();
       // wait for notification
       buffer_access.lock();
   }
```

At this point **the mutex is locked** (again) and **the condition is true.**

Spurious Wakeups

In the example code before, the loop (in the equivalent of `wait`-ing) may seem unnecessary, as the respective notification will be sent only after some action has made the condition true.

Nevertheless it makes sense and may even be necessary:

- **First of all, an implementation is allowed to give Spurious Wake-Ups, so the loop is necessary anyway.**
- If notifications are sent while nobody waits on the condition variable, it is simply discarded, therefore
  - a producer-consumer scenario is more robust if it tends to send "too many" notifications (of which some are discarded) …
  - … while sending "too few" could cause some thread to wait forever.

> Specifying the condition check in combination with `wait`-ing *does it right* and hence should be preferred over writing a loop explicitly.*

*: Thus avoiding some later maintainer considers the `while` "unnecessary" and replace it with `if` …

Atomic Operation Support

With the support for atomic operations C++11 multi-threading allows to implement

- Wait-Free Algorithms,
- Lock-Free Algorithms, and
- Obstruction-Free Algorithms.

The basic concept is to provide a way to know whether a certain modification of some memory location was caused by the current thread or by another one.

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

Atomic Operations Example

The following example, demonstrating the lock-free approach with operations (instead of using mutexes) modifies a former example:

```
template<typename T1, typename T2, typename Transformation>
void worker(T1 args[], std::size_t data_size, T2 result,
            std::atomic_size_t &total_progress,
            Transformation func) {
    while (total_progress < data_size) {
        constexpr auto chunks = std::size_t{100};
        std::size_t beg = total_progress.load();
        std::size_t end;
        do {
            end = ((data_size - beg) > chunks)
                ? beg + chunks
                : data_size;
        } while (!total_progress.compare_exchange_weak(beg, end));
        std::transform(&data[beg], &data[end], &sqrts[beg], func);
    }
}
```

Be sure to understand that the loop controlled by the return value of `compare_exchange_weak` guarantees the prior calculations are (still) valid.

Atomic Operations Recommendation

- Except for the potential of deadlocks the challenges are similar to algorithms using locks:*
  - Problems may only show for a critical timing and may be reproducible in particular test environments only.
  - In general, a failed test may show the presence of errors, but even many successful tests do not guarantee their absence.

[!] Beyond trivial cases – like the one shown in the example –, implementing multi-threaded programs with atomic operations requires **substantial expertise**.

Be sure to keep the design **as simple as possible** and have it reviewed by other developers experienced in that particular fields, maybe both, colleagues and hired consultants too.

*: It may very well be the case that problematic situations depend on hardware features like the size of cache-lines, the depth of an instruction pipeline, or the way branch prediction works.

Using Class `std::thread`

In the examples before the class `std::thread` was used only indirectly (via `std::async`, building on Futures and Promises.

- There is also a class `std::thread`
- taking any runnable code as constructor argument,
- executing it in a separate thread.

(i) For more information see:
http://en.cppreference.com/w/cpp/thread

There are explicitly no means provided to forcefully terminate one thread from another one.

[!] Any non-portable way* to forcefully terminate a thread risks to entail serious consequences later, as e.g. locks may not be released or awaited notifications not be sent.

*: Such as potentially existing *interrupt* or *kill* functions reachable via a Native Interface Handle.

Example for Using Class `std::thread`

In case worker **returns no value** and **throws no exception**, like in a former example (and assuming the same set-up), using class `std::thread` directly can be straight forward (left) or done in the "nifty" way (right):

```
// starting some workers
// multi-threaded ...
std::thread t1{worker_task};
std::thread t2{worker_task};
…
// ... and waiting for them
// to finish:
t1.join();
t2.join();
…
```

```
using namespace std;
constexpr auto NCORES = 4;
// starting one worker-thread
// per core ...
array<thread, NCORES> threads;
for (auto &t : threads)
    t = thread{worker_task};
// ... and wait for all to
// finish:
for (auto &t : threads)
    t.join();
```

An alternative to `join`-ing with a thread is `detach`-ing it.

[!] A program will immediately terminate if an instance of class `std::thread` referring to an active thread gets destructed.

Recommendations for Using Class `std::thread`

- In trivial cases (no exceptions, no result to fetch) using threads via instances of class `std::thread` may be considered.
- Nevertheless understand the peculiarities and know how to avoid race conditions, especially when `std::thread` objects go out of scope.*

[!] It causes the program to terminate if the callable started throws an exception or an instance of `std::thread` is still in *joinable state* when it goes out of scope and gets destructed.

*: At 1:00:47 the following video by Scott Meyers gives a good introduction to the problem and some recipes how it can be avoided: http://channel9.msdn.com/Events/GoingNative/2013/An-Effective-Cpp11-14-Sampler

Native Handles

Last and finally, most C++11 multi-threading implementation build on a threading model provided by their execution environment.

> The requirements in the standard are rather the intersection of the features provided by well-known threading models.

Usually and typically the standard does not mandate any any extensions thereof, but in some cases provides a way to "reach through" to the native thread model:

- E.g. `std::thread::native_handle` could provide ways to manipulate thread priorities, maybe including ways to specify protocols for Priority Ceiling or other means to circumvent Priority Inversion.
- Also the classes for condition variables and the various kinds of mutexes have member functions `native_handle`.
- The enumeration `std::launch` may provide more (named) values to control implementation specific details in the behavior of `std::async`.

Concurrency Recommendations

So far the presentation of C++11 concurrency support was only meant as an overview.

[!] Practically using concurrency features beyond parallelizing independent tasks requires much more knowledge and experience in this area, what this presentation can not provide.

A good and near to exhaustive coverage of the concurrency part of C++11 is:

C++ Concurrency in Action
Practical Multithreading
by Anthony Williams
ISBN-13: 978-1-9334988-77-1

Appendices and Info-Graphics

This sections collects:

- Info-Graphics Used Regularly
- Optional Info-Graphics

The former are linked from other pages of this presentation, the latter may be used to answer specific questions sometimes posed by participants.

Regular Info-Graphics

General Hardware Model and Mapping of C++

- Hardware Execution Model
- Class to Memory Mapping
- Exception Basic Principles
- Smart Pointers

Templates

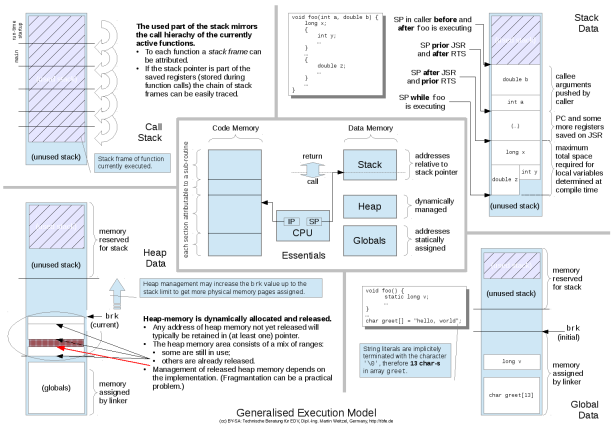
- Template Classes and Functions
- Parametrizing Types and Sizes

Standard Library

- Standard Strings
- I/O-Stream Basics
- STL Design Overview
- STL Sequence Containers
- STL Associative Containers
- C++11 Regular Expression API

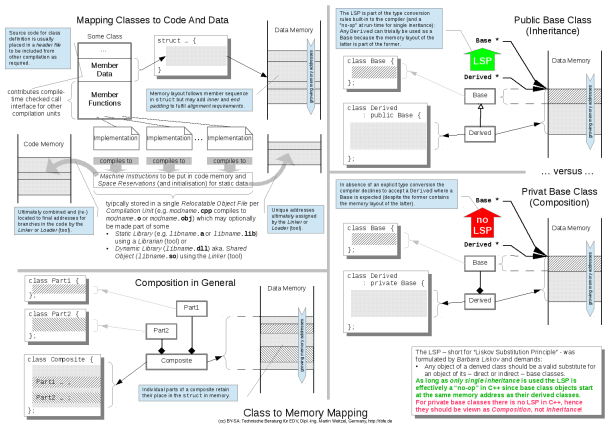
Execution Model

Basic hardware components and their typical use in C/C++ programs.



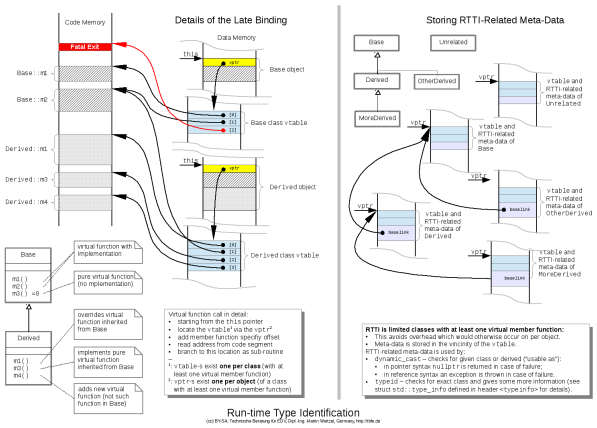
Class to Memory Mapping

How C++ classes are mapped to memory (and finally decay to C data `struct`-s), including Composition and Inheritance.



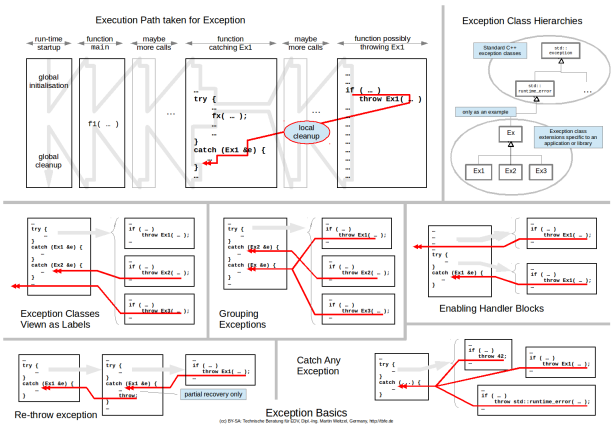
Run-Time Type Identification

Comparing specific run-time type identification via virtual member function with its explicit form.



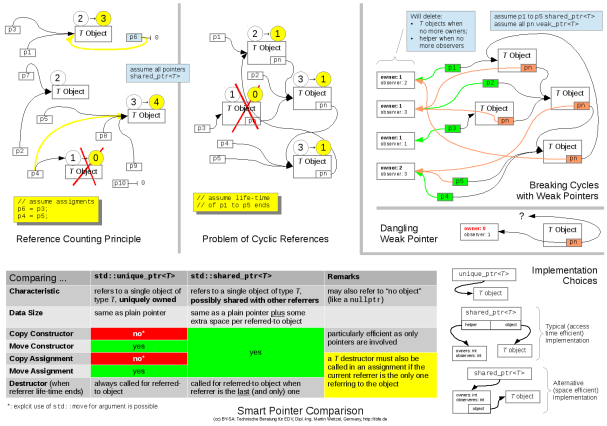
Exception Basic Principles

View of exceptions as non-local branches, typically used for error recovery paths.



C++11 Smart Pointers

Principles and Implementation of Smart Pointers (as provided by C++11).



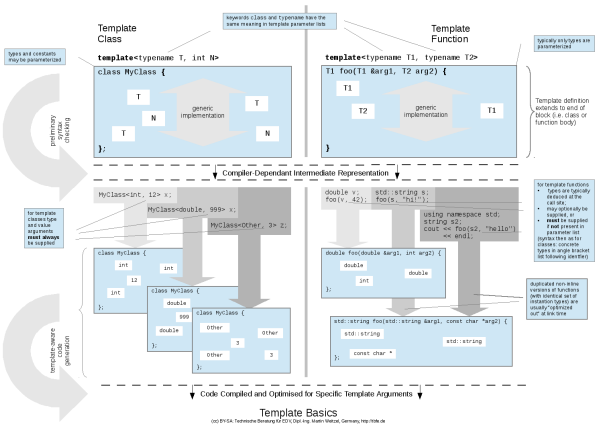
Comparing ...	std::unique_ptr<T>	std::shared_ptr<T>	Remarks
Characteristic	refers to a single object of type T, **uniquely owned**	refers to a single object of type T, **possibly shared with other referrers**	may also refer to "no object" (like a `nullptr`)
Data Size	same as plain pointer	same as a plain pointer plus some extra space per referred-to object	
Copy Constructor	no*	yes	particularly efficient as only pointers are involved
Move Constructor	yes		
Copy Assignment	no*		a T destructor must also be called in an assignment if the current referrer is the only one referring to the object
Move Assignment	yes		
Destructor (when referrer life-time ends)	always called for referred-to object	called for referred-to object when referrer is the last (and only) one	

*: explcit use of `std::move` for argument is possible

Smart Pointer Comparison

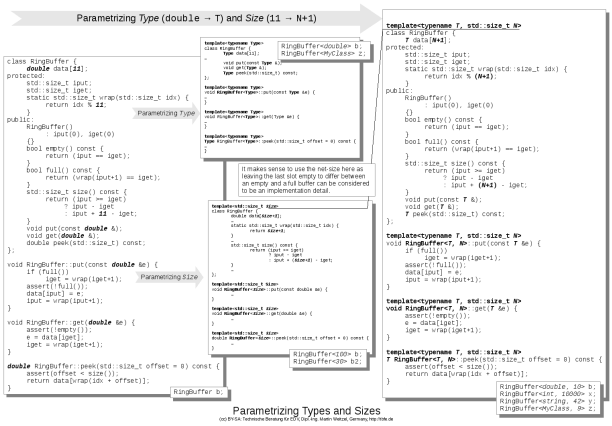
Template Classes and Functions

Basic use of C++ Templates for classes and functions.



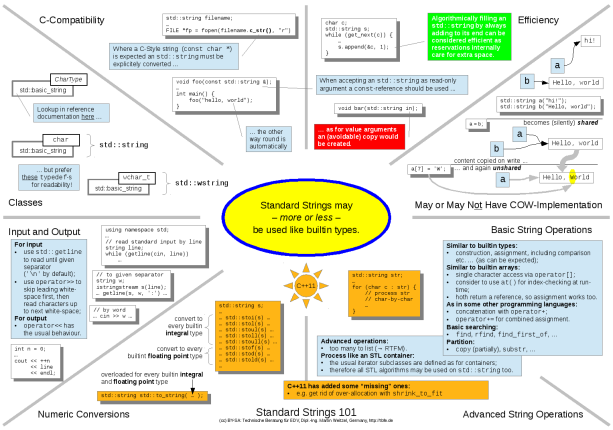
Parametrizing Types and Sizes

Example for parametrizing (element) types and size of a generic `RingBuffer` class.



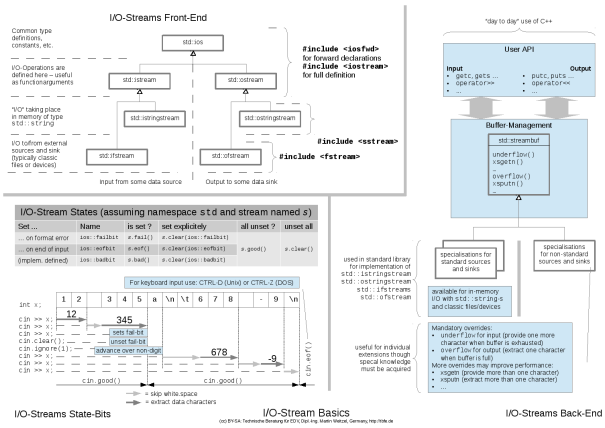
Basics String Handling

Overview of architecture and some details of the `std::string` class.



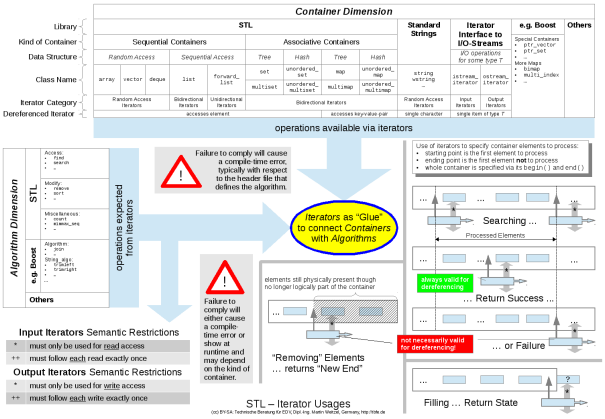
I/O-Stream Basics

Overview of architecture and some details of I/O-streams.



STL Architecture Overview

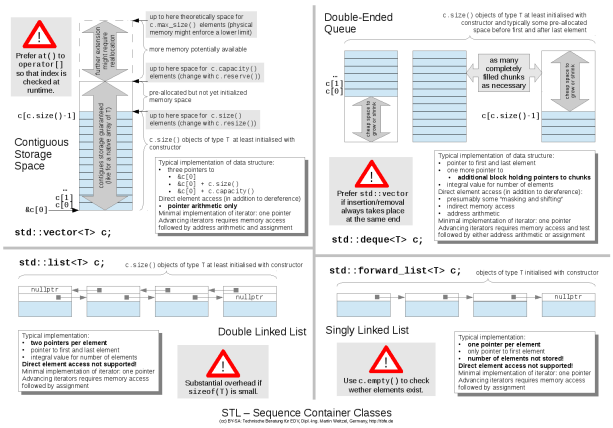
Overview of the STL Architecture, emphasizing the use of iterators as kind of "glue" between containers and algorithms



STL Sequence Containers

Depicting implementation of STL Sequence Containers.

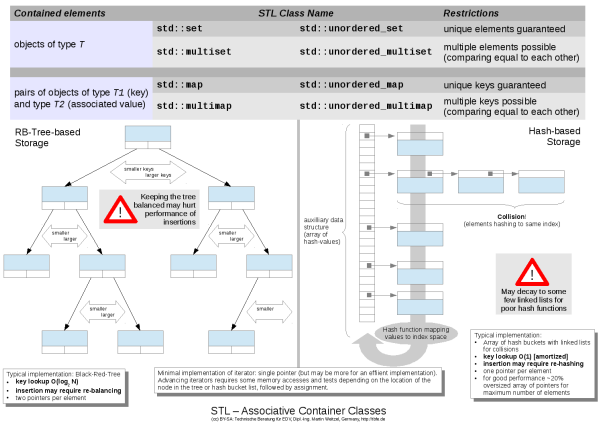
But also note: The STL specification asserts performance and other properties, **not** a specific implementation!



STL Associative Containers

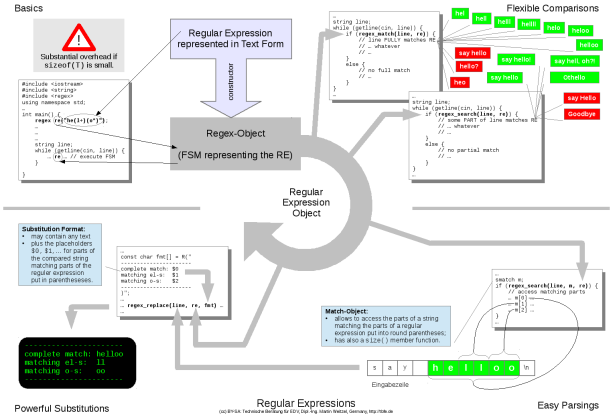
Depicting implementation of STL Associative Containers.

But also note: The STL specification asserts performance and other properties, **not** a specific implementation!



C++11 Regular Expression API

Understanding the Regular Expressions API in C++11.



Optional Info-Graphics (Mixed 1)

General Goals of Software Design

- Guiding Principles

Unified Modelling Language

- Classes and Relations in UML
- Classes Relations by Example

Various Applications of `virtual` Member Functions

- Type Based Branching (Practical RTTI)
- Two Implementations of the "Open/Close" (Principle)

Optional InfoGraphics (Mixed 2)

More on Inheritance

- Multiple Inheritance (Principle)
- Problems of Diamond-Shaped Inheritance

More on Iterators

- C++98 Iterator Categories
- Specific Iterator Details

More on Resource Management and Exceptions

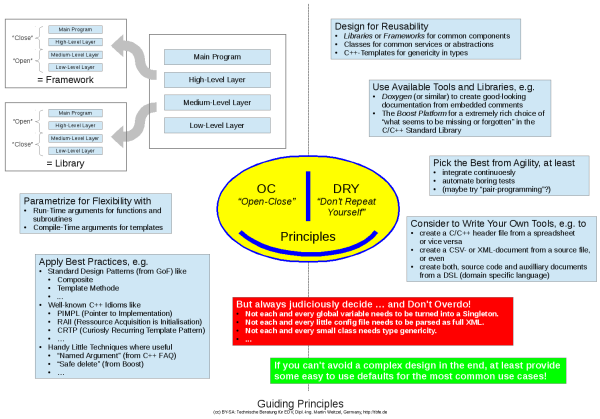
- RAII Style Resource Management
- Specific Exception Details

More on C++ Templates

- Reducing Code Bloat by Templates

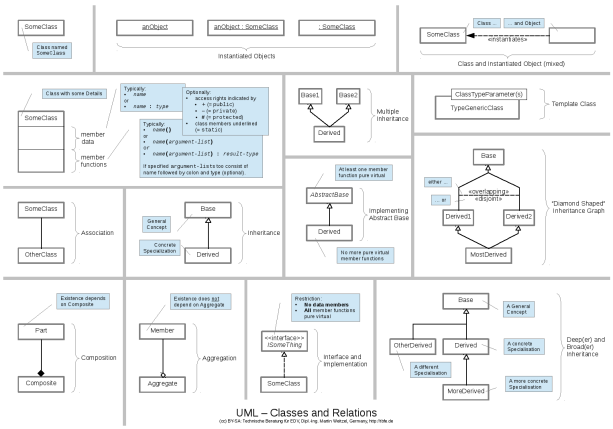
Guiding Principles

Partitioning to tackle complexity and provide chances to come up with to components that can be used in flexible ways is at the core of all software design (if not at the core of all engineering).



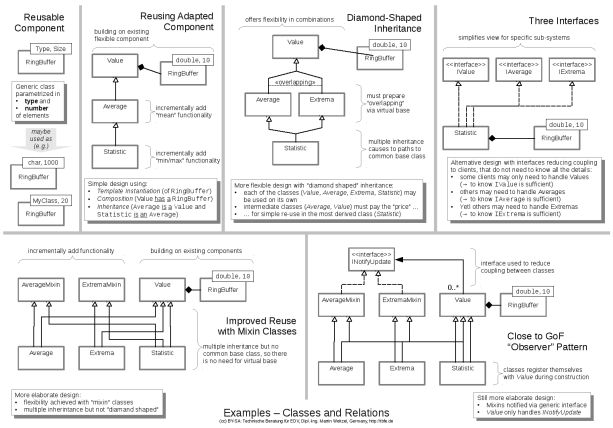
Classes and Relations (Quick UML)

UML Notation to graphically depict classes and their typical relations.



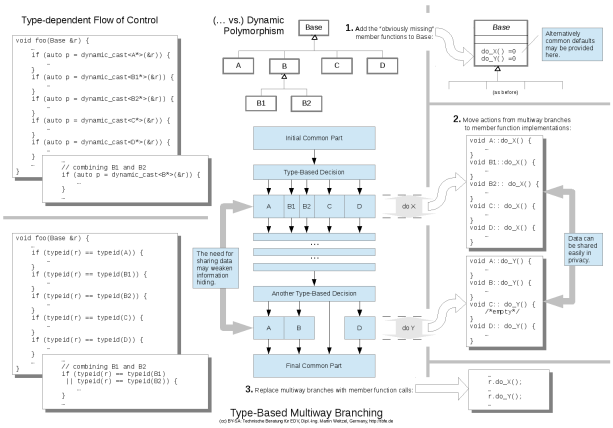
Class Relations by Example

Some examples for class relations, highlighting options for reuse and flexible extensibility.



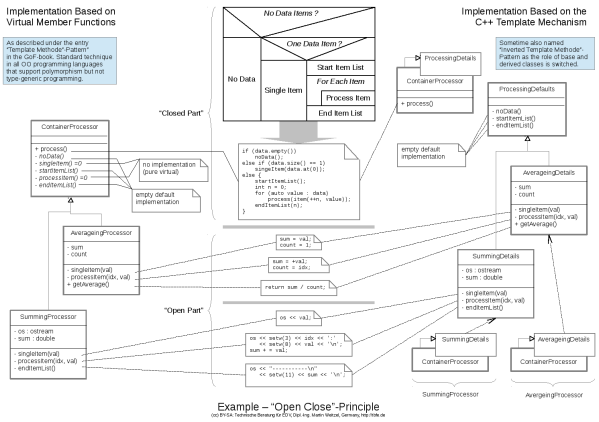
Type Based Branching

Comparing explicit use of RTTI to (usually more elegant) type-based branching via virtual member functions.



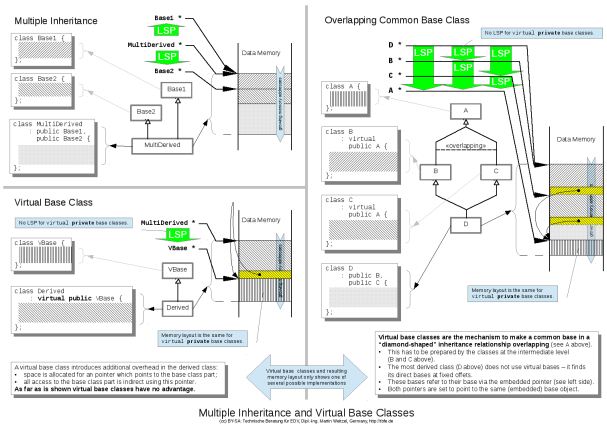
Two Implementations of "Open/Close"

Comparing the "open/close" approach based on virtual member functions and templates.



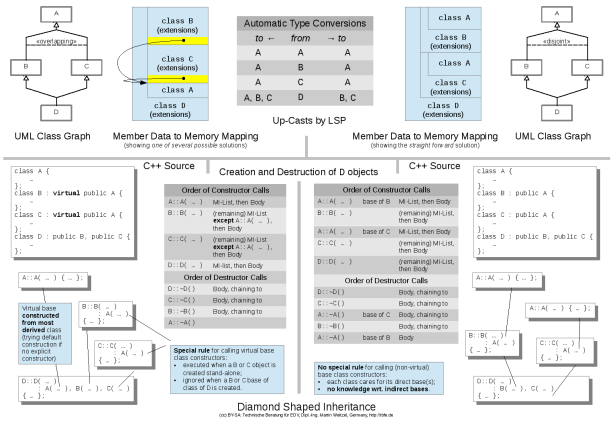
Mutiple Inheritance (Principle)

Showing the principles of multiple inheritance (also preparing diamond-shaped inheritance).



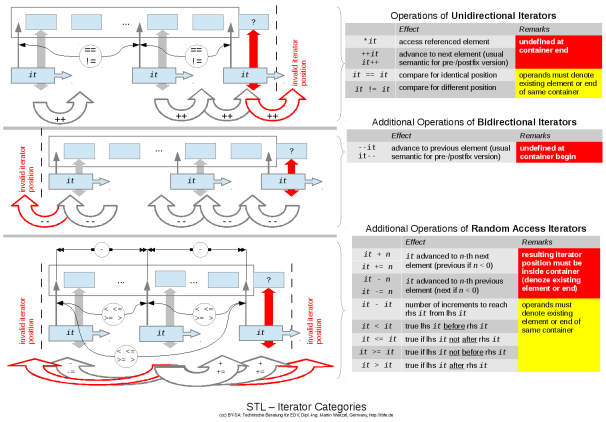
Diamond-Shaped Inheritance

Showing special handling and considerations if multiple inheritance turns into a diamond-shaped relation.



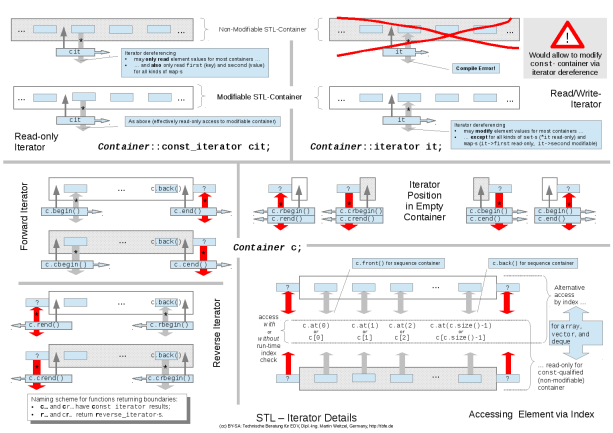
C++98 Iterator Categories

Main Iterator Categories as defined by C++98.



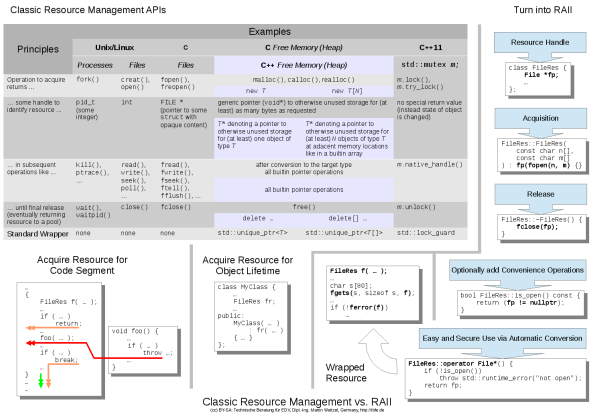
IteratorSpecific Details

Some more iterator specific details (like `const_`-iterators versus mutable ...).



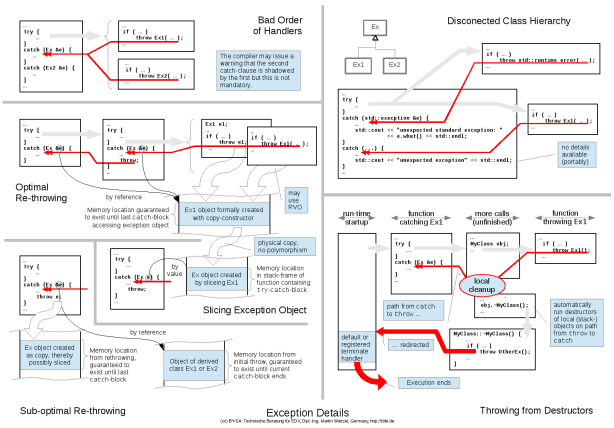
RAII Style Resource Management

Managing Resources via RAII (Resource Acquisition is Initialisation).



Specific Exception Details

Some more specific details on exceptions (like *do*-s and *don't*-s).



Reducing Code Bloat by Templates

Understanding the nature of C++ templates and how to reduce code bloat.