ACCU 2017

C++ CORE GUIDELINES - SAFER CODE

Modernize your C++ Code Base

IFS INSTITUTE FOR SOFTWARE

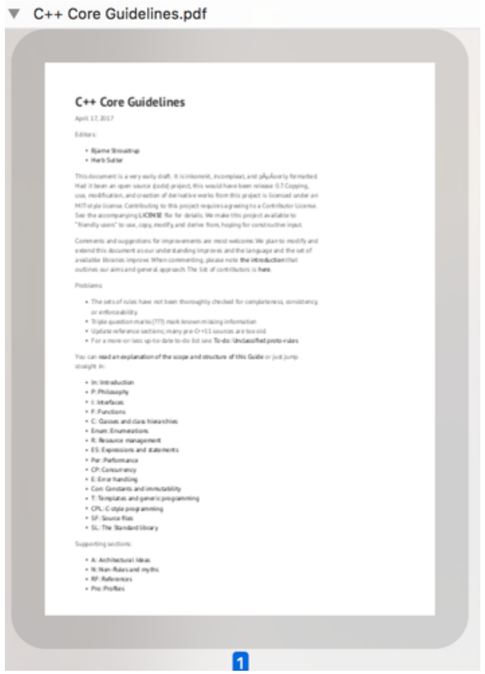
Prof. Peter Sommerlad
Director of IFS

April 2017

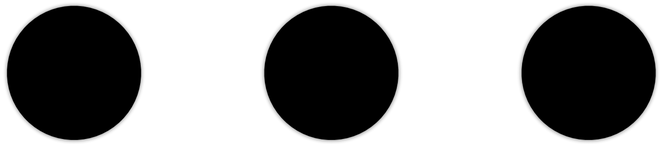
HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL

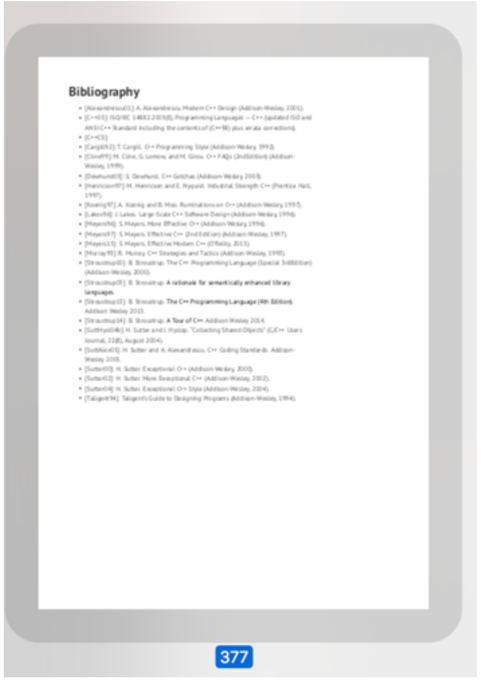
FHO Fachhochschule Ostschweiz

Core?



page 1





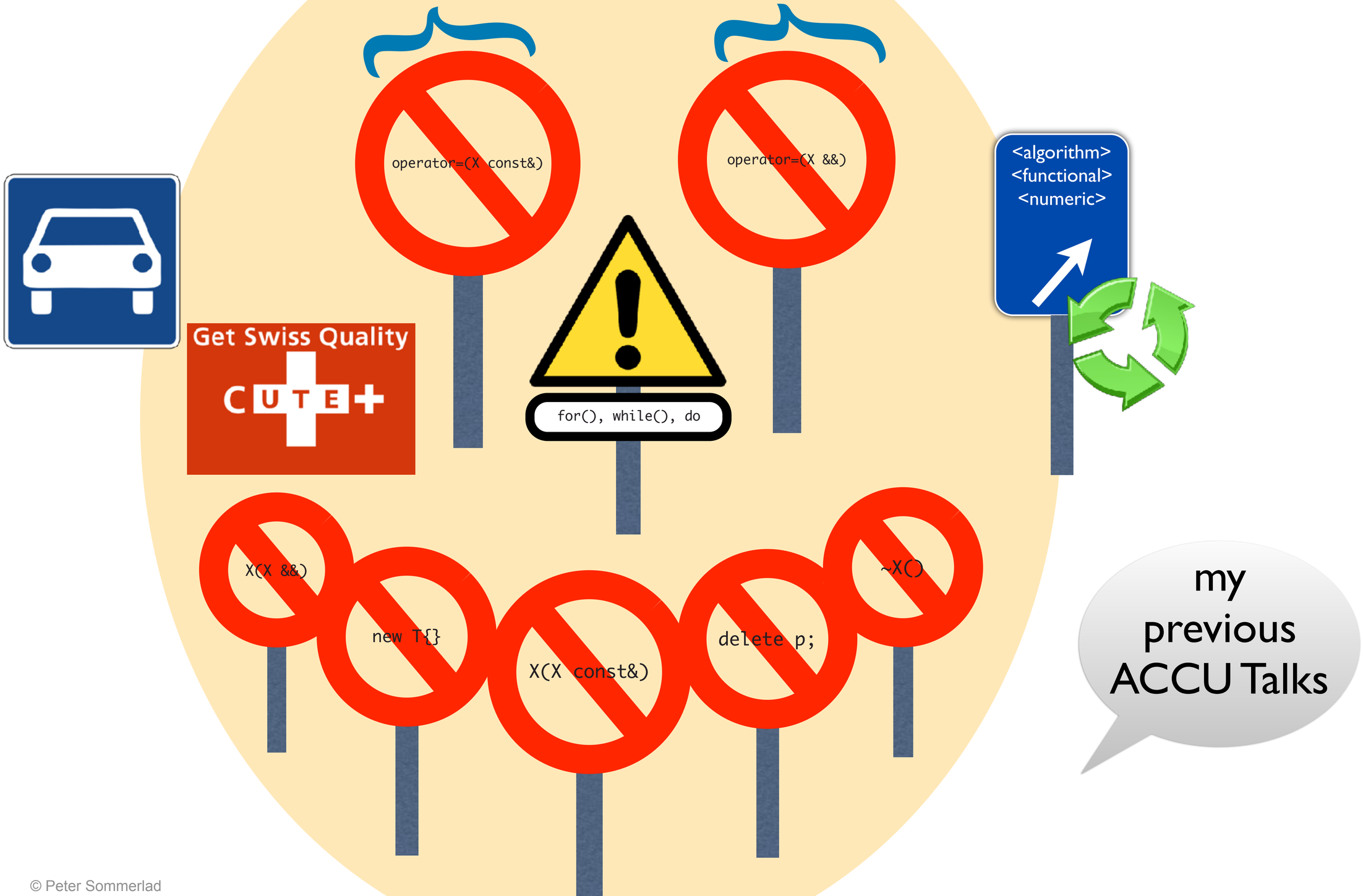
page 377

Guide-lines?

C++ Core Guidelines

work in progress, not finished

- **Goal: transition “legacy” C++ code towards modern C++**
  - but which might not be followed by others or in older code
- **underlying idea: provide static analysis tools to warn about some violations**
  - Cevelop already provides some of the corresponding checkers
- **Philosophy is to write modern standard C++ code**
  - express intent in the language not comments (P.S.: “Only the code tells the truth”)
  - use good naming
  - know the standard library and libraries you actually use
  - employ the type system: name types and abstractions - sidestep simple types where appropriate
  - type safety and compile-time checking - run time errors if needed but always checked for, no UB or leaks
- **Supported by GSL - Guideline Support Library**
  - RAII with finally, Safe Narrowing, Contracts, span<T>, string_span, pointer stuff, byte
  - header-only, no linking, portable



Some random C++ Core Guidelines Examples

ES.1: Prefer the standard library to other libraries and to "handcrafted code"

Reason Code using a library can be much easier to write than code working directly with language features, much shorter, tend to be of a higher level of abstraction, and the library code is presumably already tested. The ISO C++ standard library is among the most widely known and best tested libraries. It is available as part of all C++ Implementations.

Example

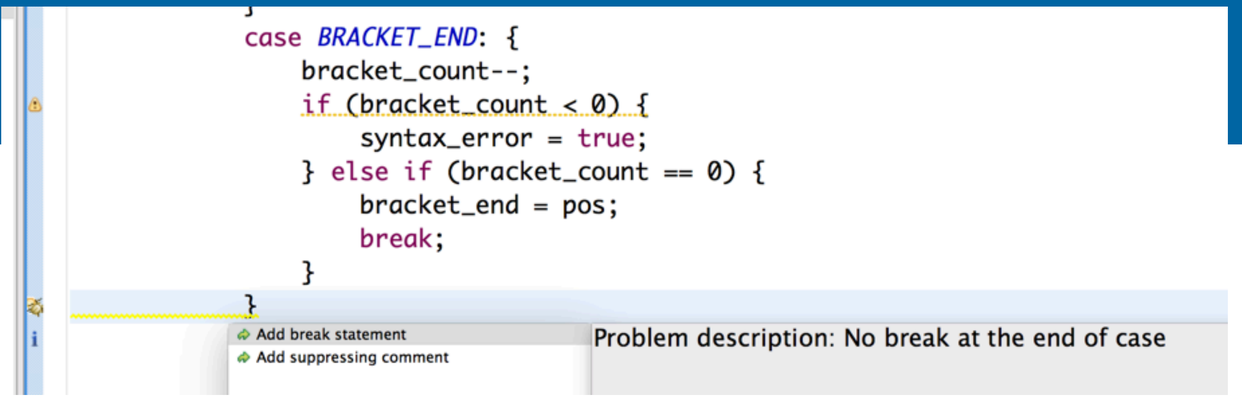
```
auto sum = accumulate(begin(a), end(a), 0.0);   // good
```

a range version of `accumulate` would be even better:

```
auto sum = accumulate(v, 0.0); // better
```

but don't hand-code a well-known algorithm:

```
int max = v.size();   // bad: verbose, purpose unstated
double sum = 0.0;
for (int i = 0; i < max; ++i)
    sum = sum + v[i];
```



ES.78: Always end a non-empty `case` with a `break`

Reason Accidentally leaving out a `break` is a fairly common bug. A deliberate fallthrough is a maintenance hazard.**Example**

```
switch (eventType)
{
case Information:
    update_status_bar();
    break;
case Warning:
    write_event_log();
case Error:
    display_error_window(); // Bad
    break;
}
```

...

Note Multiple case labels of a single statement is OK:

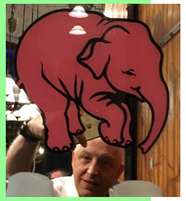
```
switch (x) {
case 'a':
case 'b':
case 'f':
    do_something(x);
    break;
}
```

Enforcement Flag all fallthroughs from non-empty `cases`.

My take on the Core Guidelines

Pros

- Fosters Modern C++ Style
- Safer Code - less Undefined behavior
- Pointer Safety
- Resource Management
- Parameter Passing
- Good Software Engineering Principles
- Less Verbosity
- Common Sense (which might not be so common)
- Rid code of “C-isms” and 1990s C++
- Provide transformation guidelines
- Helper Library (GSL)
- Potential for static analysis checks



Cons

- Too many rules, can’t know them all
- Rules must be prioritized to be useful
- Some rules only provide bad examples
- Overlap in Rules
- Categorization not always clear
- Some rules can not be adapted incrementally without losing effectiveness
- Common sense
- Specialist rules, you should not write code that needs them, unless you should already know what you are doing
- Too modern for your environment
- C++17/20 will make some helpers obsolete
- lacks opposite of owner<T>

C++ Core Guidelines: Philosophy Guidelines P.1 .. P.11

1. **Express ideas directly in code**
2. **Write in ISO Standard C++**
3. **Express intent**
4. **Ideally, a program should be statically type safe**
5. **Prefer compile-time checking to run-time checking**
6. **What cannot be checked at compile time should be checkable at run time**
7. **Catch run-time errors early**
8. **Don't leak any resources**
9. **Don't waste time or space**
10. **Prefer immutable data to mutable data**
11. **Encapsulate messy constructs, rather than spreading through the code**

C++ Core Guidelines: Express Ideas Directly in Code

1. **Express ideas directly in code**
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- Comments are not compiled
  - name functions, types, variables accordingly
- Apply the “Whole Object” Pattern
  - types for units, UDL for constants, not double
- Avoid self-written loops in favor of algorithms

```
// no comment..
// Example
auto g= 9.81_m/(1s*1s);
```

see my talk from ACCU 2016 on Units:
https://www.youtube.com/watch?v=N94oNLVNyLM

C++ Core Guidelines: Write in ISO Standard C++

1. Express ideas directly in code
2. **Write in ISO Standard C++**
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- Compilers tend to be too generous
- Beware platform dependency (esp. MS)
- Beware compiler extensions silently enabled (gcc)
- Use multiple compilers (clang and gcc)
- Code might not port to more modern standards

```
for example:

g++ -std=c++14 -pedantic-errors -Werr -Wall -Wextra

or

g++ -std=c++17 -pedantic-errors -Werr -Wall -Wextra

//my take: sidestep #define macros
// -> C++ constexpr functions transformation
// -> Macronator: macro inlining as last resort
```

C++ Core Guidelines: Express Intent

1. Express ideas directly in code
2. Write in ISO Standard C++
3. **Express intent**
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- see P.1 express ideas directly in code
- Use range-for loop or better algorithms
  - instead of while with external loop variables or for using counters/iterators explicitly
- Know the language and the standard library!

```
int i = 0;
while (i < v.size()) {
    // ... do something with v[i] ...
}

// better
for(auto const &x:v){
    //  ... do something with x ...
}

// or
for_each(begin(v),end(v),[](auto const & x){
    //  ... do something with x ...
});
```

C++ Core Guidelines: A Program should be Statically Type Safe

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. **Ideally, a program should be statically type safe**
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- ~~unions~~ - use variant (C++17/boost)
- ~~casts~~ - every cast denotes a design problem
  - very few exceptions in library/low-level code
- array to pointer decay, range errors
  - use span (C++17/GSL), array<>, or string_view
- narrowing conversions - GSL narrow_cast, {init}

```
variant<uchar8_t,uint16_t,uint32_t,uint64_t>

double sum(double *da, size_t n) //->
double sum(span<double> da)

int const i{42.2}; // compile error, vs. int i=42.2;
```

C++ Core Guidelines: Prefer Compile-time Checking to Run-time Errors

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. **Prefer compile-time checking to run-time checking**
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- use -Werr etc. see P.2
- run static analysis tools (Cevelop, Linticator, etc)
- static_assert to check compile-time assumptions
  - e.g. bit sizes
- use gsl::span<T> to avoid size errors in functions using arrays

```
static_assert(sizeof(int)==4, “must run on 32bit machine”);
static_assert(std::is_signed_v<char>); // C++17
static_assert(std::is_signed<char>::value,”char must be signed”);

static_assert(sizeof(void*)==sizeof(int), “pointer size wrong”);
```

C++ Core Guidelines: Checkable at Run-time

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. **What cannot be checked at compile time should be checkable at run time**
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- core guidelines explanation is weak for this topic
- sidestep pointers
  - use smart pointers and make_xxx functions to manage memory
- C++20 might come with “Contracts” support
  - check pre- and post-conditions!
  - GSL provides gsl_assert library for contracts
- Conscious error and exception handling!

```
auto pint=make_unique<int>(42);
auto dynintarray=make_unique<int[]>(100);
shared_ptr<base> p=make_shared<subclass>(ctor_arguments);


array<int,10> a;

a.at(42); // throws
```

C++ Core Guidelines: Catch run-time errors early

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. **Catch run-time errors early**
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- again, a weakly described topic
- Do range checks early, e.g., use at() instead of []
  - better avoid the need for range checks
  - use vector, span, range-for, algorithms

```
double sum(double *da, size_t n) //->
double sum(span<double> da)
//or just use a vector with
accumulate(begin(v),end(v),0);
```

C++ Core Guidelines: Don’t leak any resources

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. **Don't leak any resources**
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

- RAII - resource acquisition is initialization
- unique_ptr and make_unique
- shared_ptr and make_shared
- scope guards: lock_guard, unique_lock
- C++20?: scope_guard, unique_resource
- NO NAKED OWNING POINTERS
- NO explicit new/delete/malloc/free/fopen/strdup etc
- USE std:: vector, string, array instead of pointers

```
auto pi=make_unique<int>(6*7);

auto guard=gsl::finally([]{std::cout << “cleanup”;});

// for C resources
auto s=unique_ptr<char const,void(*)(void *)>
           {strdup(“hello”),&::free};
auto f=unique_ptr<FILE,decltype(::fclose)>
           {fopen(“hello.txt”,”r”),::fclose}; // use ifstream!
```

C++ Core Guidelines: P.9 Don’t waste time or space

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. **Don't waste time or space**
10. Prefer immutable data to mutable data
11. Encapsulate messy constructs, rather than spreading through the code

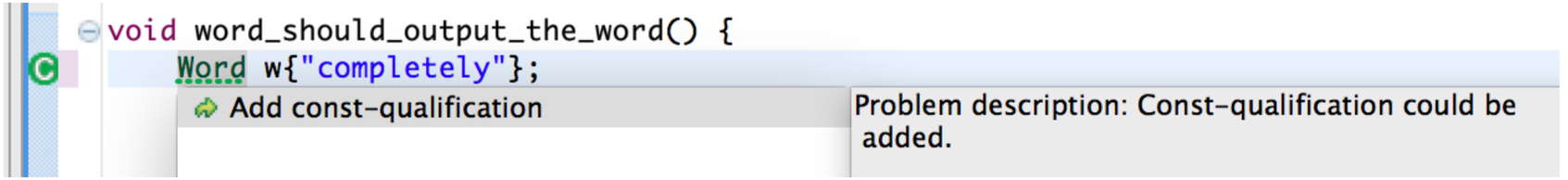
- This includes space for source code
  - implies time to understand it, see P.1
- Learn the Standard Library!
  - especially: vector, string, array, map
  - and the algorithms

```
// seen things like that in production code!
vector<int> v1(10);
for (int i=0; i < v1.size(); ++i)
    v1[i]=42;
vector<int> v2;
for (vector<int>::iterator it=v1.begin(); it != v1.end();++it)
    v2.push_back(vi[it-v1.begin()]);

// better
vector<int> v1(10,42); // need to use () instead of {}
auto v2=v1; // or vector<int> v2{v1};
```

C++ Core Guidelines: Prefer immutable data to mutable data

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. **Prefer immutable data to mutable data**
11. Encapsulate messy constructs, rather than spreading through the code

- make your code
  - “As const as possible, but not more” (Dewhurst)
- Cevelop provides the “Constificator” plug-in automating const introduction
- Few exceptions might lead to less efficient code
  - NRVO, mutated parameters passed by value
- remember const value parameters do not influence overload resolution




```
void word_should_output_the_word() {
    Word const w { "completely" };
    std::ostringstream output{};
    output << w;
    ASSERT_EQUAL(w.str(), output.str());
}
```

C++ Core Guidelines: P.11 Encapsulate messy constructs

1. Express ideas directly in code
2. Write in ISO Standard C++
3. Express intent
4. Ideally, a program should be statically type safe
5. Prefer compile-time checking to run-time checking
6. What cannot be checked at compile time should be checkable at run time
7. Catch run-time errors early
8. Don't leak any resources
9. Don't waste time or space
10. Prefer immutable data to mutable data
11. **Encapsulate messy constructs, rather than spreading through the code**

- Use the available abstractions correctly
- Provide abstractions for your domain
- If it looks ugly, encapsulate
- Remember: “Less code == more software” — Kevlin Henney.

```
Extract function refactoring
currently updated by master student. (next CDT release?)

Type and Template aliases with using.

Extract Template Parameter refactoring for generalizing of
code.

For future simpler meta-programming ideas, see boost.hana
library and some ACCU 2017 talks!
```

C++ Core Guidelines Areas Overview (no time for 377 pages, some examples)

- **Interfaces**
- **Functions**
- **Classes and Hierarchies**
- **Enumerations**
- **Resource Management**
- **Expressions and Statements**
- **Concurrency and Parallelism**
- **Error Handling**
- **Constants**
- **Templates**
- **C-style Programming**
- **Source Files**
- **Standard Library**

- **Supporting sections**
  - Architecture
  - Non-rules and Myths
    - ~~all declarations on top of function~~
    - ~~single-return rule~~
    - ~~no exceptions~~
    - ~~one class per source file~~
    - ~~two-phase initialization~~
    - ~~goto exit~~
  - References
  - Profiles
  - Guideline Support Library (GSL)
  - Naming and Layout

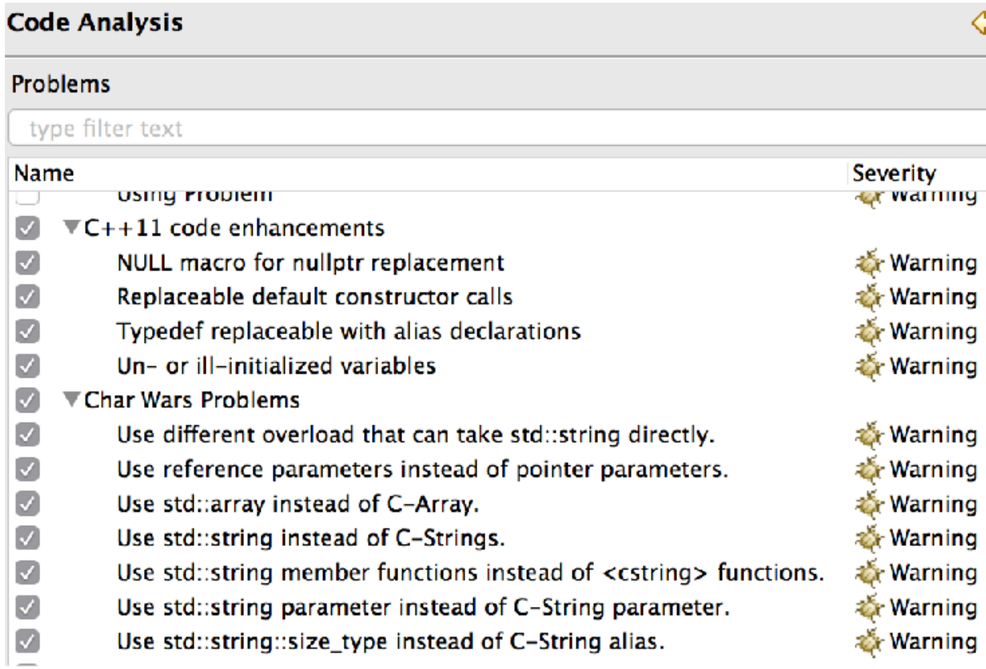
C++ Core Guidelines: Some Problem Areas addressed

- **Resource Leaks**
  - solution: smart pointers, RAII classes, ownership
- **Using invalid Pointers (dangling, casts)**
  - solution: no raw pointers + much more
- **Memory corruption**
  - bounds checks, avoid dangling pointers
- **Type System circumvention through casts, void *, etc.**
  - solution: employ static type safety
  - rid code of C-style casts
- **Code understandability**
  - solution: suggest syntax from different choices, good naming, sidestep traps

```
Pointers: prefer references

Single Object: T *          we: borrower<T*>

Parameter:     (T*, size_t n) -> (span<T>)

Ownership of memory: owner<T*> p_owner=new T;
Must delete, better unique_ptr<T> and make_unique

Shared ownership: shared_ptr<T>

Arrays: std::array<T,n> for fixed size
        std::vector<T> for run-time sized
if required: learn about C++ allocators for
specific embedded needs and parameterize vector
with it.

Absolutely NO POINTER ARITHMETIC!!!
```

Resource Management - Ownership

- **Core Guidelines: Use raw pointers only to denote non-owning single object pointers**
  - no array, no ownership, can be nullptr.
    We suggest using borrower<T> for that to enable code base migration
- **Principle: manage memory with smart pointers or mark “owning” pointers**
  - a “naked” raw pointer never owns the memory
    -> no delete p if p is a naked pointer
- **template <typename T> using owner=T; // in gsl**
  - semantic-free syntactic marker, requires dedicated static analysis tool
  - missing: observer<T>/borrower<T> to mark legally “naked” pointers as well during transition
- **template <typename T> using borrower=T; // in our gsl extension for code migration**
  - standard will provide observer_ptr<T> with a bit more semantics (Library Fundamentals TS v2).
- **Future Cevelop Releases should include more pointer modernization refactorings**
  - currently char* to std::string, plain arrays to std::array<>



A comment on Ownership, esp. owner<T*>

- **gsl::owner<T> is a transient interim solution, better use RAII**
  - owner<T> marks self-managed resources that require a destructor
- **C++20 will contain std::unique_resource<> for non-pointer resources RAII**
- **I should propose a specialization of std::unique_ptr that works for C-pointers without overhead**
  - std::unique_ptr<char *,decltype(&::free)> needs to store &free per pointer
  - This should go into GSL as well…
  - POSIX will be around for some time…
- **I came up with it 25.04.2017,**
  - when preparing the slides :-)

```
template <typename T>
struct default_free{
   void operator()(T *p) const {
      ::free(const_cast<std::remove_const_t<T>*>(p));
   }
};
template <typename T>
using unique_C_ptr=std::unique_ptr<T,default_C_freer<T>>;

static_assert(sizeof(char *)==
              sizeof(unique_C_ptr<char>),"");
```

C++ Core Guidelines: Interfaces

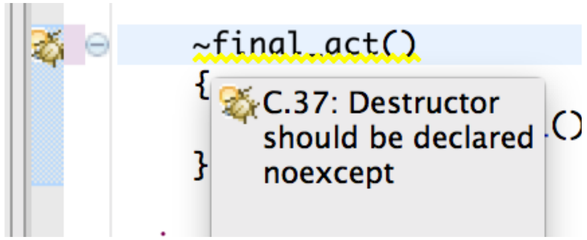
- **Explicit Interfaces**
- **No global variables**
- **No singletons**
- **Precise and strongly typed interfaces**
- **Preconditions & Postconditions**
  - (gsl: Expects(cond) and Ensures(cond))
- **state template parameters with concepts**
  - //requires until compilers can do
- **Use exceptions for signaling failure**
- **No ownership transfer via raw T***
- **non-nullable pointers with gsl::not_null<T>**
  - better consider using references
- **no array decay on interfaces**
- **no complex global initialization at run-time**
- **stick to only few parameters per function**
- **no unrelated parameters of same type**
  - doit(bool,bool,bool) is very bad!
- **abstract classes as interfaces to hierarchies**
- **cross-compiler ABI should stick to C-style**
  - see last week Hourglass Interfaces

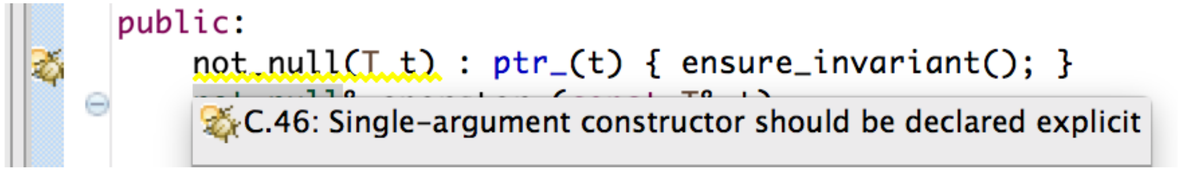
Function rules

- **Function definition rules:**
  - F.1: "Package" meaningful operations as carefully named functions
  - F.2: A function should perform a single logical operation
  - F.3: Keep functions short and simple
  - F.4: If a function may have to be evaluated at compile time, declare it constexpr
  - F.5: If a function is very small and time-critical, declare it inline
  - F.6: If your function may not throw, declare it noexcept
  - F.7: For general use, take T* or T& arguments rather than smart pointers
  - F.8: Prefer pure functions
- **Parameter passing expression rules:**
  - F.15: Prefer simple and conventional ways of passing information
  - F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const
  - F.17: For "in-out" parameters, pass by reference to non-const
  - F.18: For "consume" parameters, pass by X&& and std::move the parameter
  - F.19: For "forward" parameters, pass by TP&& and only std::forward the parameter
  - F.20: For "out" output values, prefer return values to output parameters
  - F.21: To return multiple "out" values, prefer returning a tuple or struct
  - F.60: Prefer T* over T& when "no argument" is a valid option
- **Parameter passing semantic rules:**
  - F.22: Use T* or owner<T*> or a smart pointer to designate a single object
  - F.23: Use a not_null<T> to indicate "null" is not a valid value
  - F.24: Use a span<T> or a span_p<T> to designate a half-open sequence
  - F.25: Use a zstring or a not_null<zstring> to designate a C-style string
  - F.26: Use a unique_ptr<T> to transfer ownership where a pointer is needed
  - F.27: Use a shared_ptr<T> to share ownership
- **Value return semantic rules:**
  - F.42: Return a T* to indicate a position (only)
  - F.43: Never (directly or indirectly) return a pointer to a local object
  - F.44: Return a T& when copy is undesirable and "returning no object" isn't an option
  - F.45: Don't return a T&&
  - F.46: int is the return type for main()
  - F.47: Return T& from assignment operators.
- **Other function rules:**
  - F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function)
  - F.51: Where there is a choice, prefer default arguments over overloading
  - F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms
  - F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread
  - F.54: If you capture this, capture all variables explicitly (no default capture)

Classes and Hierarchies (I am not satisfied with all of those)

- C.1: Organize related data into structures (structs or classes)
- *C.2: Use class if the class has an invariant; use struct if the data members can vary independently*
- C.3: Represent the distinction between an interface and an implementation using a class
- C.4: Make a function a member only if it needs direct access to the representation of a class
- **C.5: Place helper functions in the same namespace as the class they support**
- C.7: Don't define a class or enum and declare a variable of its type in the same statement
- C.8: use class rather that struct if any member is non-public
- **C.9: minimize exposure of members**





Subsections:

- C.concrete: Concrete types
- **C.ctor: Constructors, assignments, and destructors**
- C.con: Containers and other resource handles
- C.lambdas: Function objects and lambdas
- **C.hier: Class hierarchies (OOP)**
- **C.over: Overloading and overloaded operators**
- **C.union: Unions**

Default class operations rules (includes Rule of Zero)

- **Set of default operations rules**
  - C.20: If you can avoid defining any default operations, do
  - C.21: If you define or =delete any default operation, define or =delete them all
  - C.22: Make default operations consistent
- **Destructor rules:**
  - C.30: Define a destructor if a class needs an explicit action at object destruction
  - C.31: All resources acquired by a class must be released by the class's destructor
  - C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning
  - C.33: If a class has an owning pointer member, define or =delete a destructor
  - C.34: If a class has an owning reference member, define or =delete a destructor
  - C.35: A base class with a virtual function needs a virtual destructor
  - C.36: A destructor may not fail
  - C.37: Make destructors noexcept
- **Constructor rules:**
  - C.40: Define a constructor if a class has an invariant
  - C.41: A constructor should create a fully initialized object
  - C.42: If a constructor cannot construct a valid object, throw an exception
  - C.43: Ensure that a class has a default constructor
  - C.44: Prefer default constructors to be simple and non-throwing
  - C.45: Don't define a default constructor that only initializes data members; use member initializers instead
  - C.46: By default, declare single-argument constructors explicit
  - C.47: Define and initialize member variables in the order of member declaration
  - C.48: Prefer in-class initializers to member initializers in constructors for constant initializers
  - C.49: Prefer initialization to assignment in constructors
  - C.50: Use a factory function if you need "virtual behavior" during initialization
  - C.51: Use delegating constructors to represent common actions for all constructors of a class
  - C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization
- **Copy and move rules:**
  - C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&
  - C.61: A copy operation should copy
  - C.62: Make copy assignment safe for self-assignment
  - C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&
  - C.64: A move operation should move and leave its source in a valid state
  - C.65: Make move assignment safe for self-assignment
  - C.66: Make move operations noexcept
  - C.67: A base class should suppress copying, and provide a virtual clone instead if "copying" is desired
- **Other default operations rules:**
  - C.80: Use =default if you have to be explicit about using the default semantics
  - C.81: Use =delete when you want to disable default behavior (without wanting an alternative)
  - C.82: Don't call virtual functions in constructors and destructors
  - C.83: For value-like types, consider providing a noexcept swap function
  - C.84: A swap may not fail
  - C.85: Make swap noexcept
  - C.86: Make == symmetric with respect of operand types and noexcept
  - C.87: Beware of == on base classes
  - C.89: Make a hash noexcept

Resource Management ~> use smart pointers

- **Resource management rule summary:**
  - R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)
  - R.2: In interfaces, use raw pointers to denote individual objects (only)
  - R.3: A raw pointer (a T*) is non-owning
  - R.4: A raw reference (a T&) is non-owning
  - R.5: Prefer scoped objects
  - R.6: Avoid non-const global variables
- **Allocation and deallocation rule summary:**
  - R.10: Avoid malloc() and free()
  - R.11: Avoid calling new and delete explicitly
  - R.12: Immediately give the result of an explicit resource allocation to a manager object
  - R.13: Perform at most one explicit resource allocation in a single expression statement
  - *R.14: ??? array vs. pointer parameter*
  - R.15: Always overload matched allocation/ deallocation pairs
- **Smart pointer rule summary:**
  - R.20: Use unique_ptr or shared_ptr to represent ownership
  - R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership
  - R.22: Use make_shared() to make shared_ptrs
  - R.23: Use make_unique() to make unique_ptrs
  - R.24: Use std::weak_ptr to break cycles of shared_ptrs
  - R.30: Take smart pointers as parameters only to explicitly express lifetime semantics
  - R.31: If you have non-std smart pointers, follow the basic pattern from std
  - R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget
  - R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget
  - R.34: Take a shared_ptr<widget> parameter to express that a function is part owner
  - R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer
  - *R.36: Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object ???*
  - R.37: Do not pass a pointer or reference obtained from an aliased smart pointer

Naming and Layout

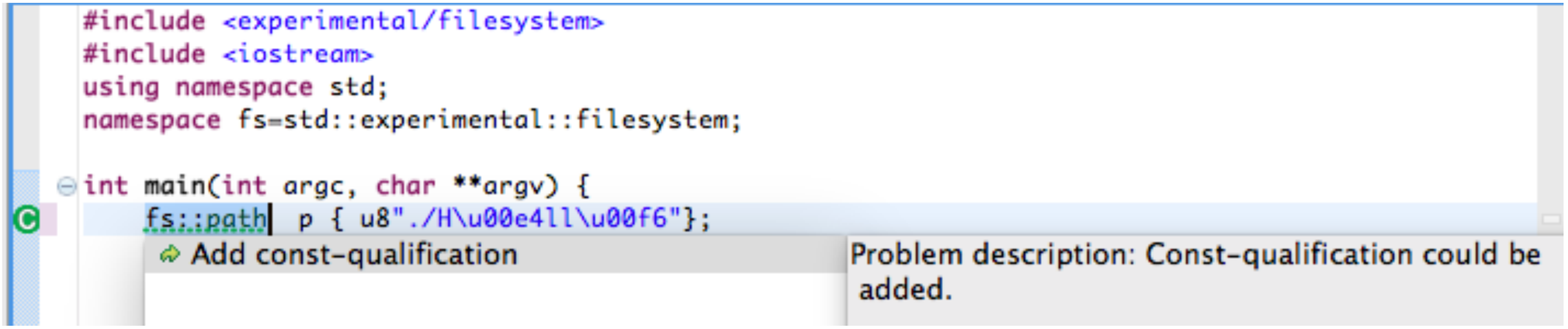
- **NL 1: Don't say in comments what can be clearly stated in code**
- **NL.2: State intent in comments**
- **NL.3: Keep comments crisp**
- **NL.4: Maintain a consistent indentation style**
- **NL.5: Don't encode type information in names (aka Hungarian Notation)**
- **NL.7: Make the length of a name roughly proportional to the length of its scope**
- **NL.8: Use a consistent naming style**
- **NL 9: Use ALL_CAPS for macro names only**
- **NL.10: Avoid CamelCase**
- **NL.15: Use spaces sparingly**
- **NL.16: Use a conventional class member declaration order**
- **NL.17: Use K&R-derived layout**
- **NL.18: Use C++-style declarator layout**
- **NL.25: Don't use void as an argument type**

```
// OTCTTT

// better use better names, IMHO

// read: no comments needed!


// using an IDE makes that automatic

// thank you MS


double foo(int x)
{
    if (0 < x) {

// get rid of them with MACRONATOR
```

```
T& operator[](size_t);   // OK
T &operator[](size_t);   // just strange
T & operator[](size_t);   // undecided

void f(void);   // bad
void g();       // better
```

Con: Constants and Immutability

- **Constant rule summary:**
  - Con.1: By default, make objects immutable
  - Con.2: By default, make member functions const
  - Con.3: By default, pass pointers and references to consts
  - Con.4: Use const to define objects with values that do not change after construction
  - Con.5: Use constexpr for values that can be computed at compile time
- Con.1-4 are already enforced by Constificator in Cevelop
  - thanks to Felix Morgner, Benjamin Gächter and Mario Meili

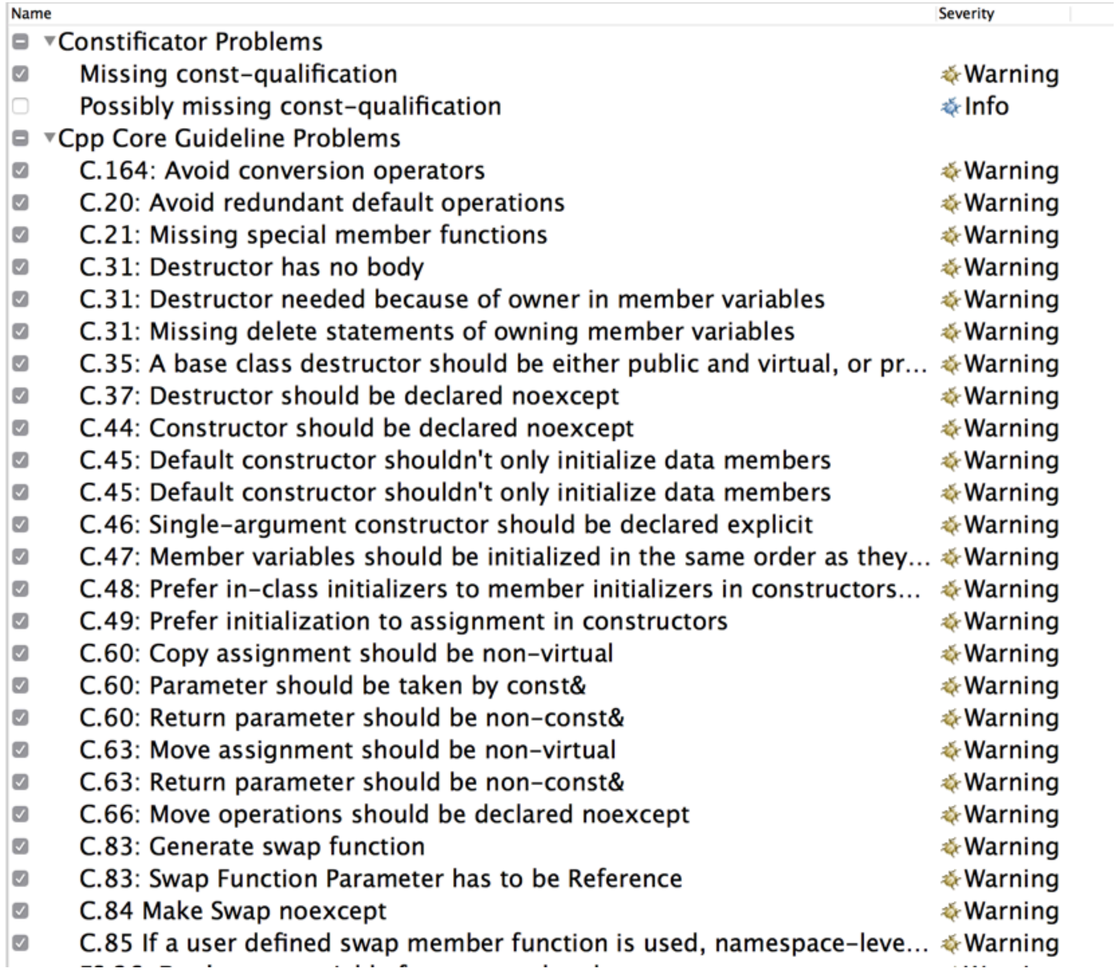


Guideline Support Library: GSL

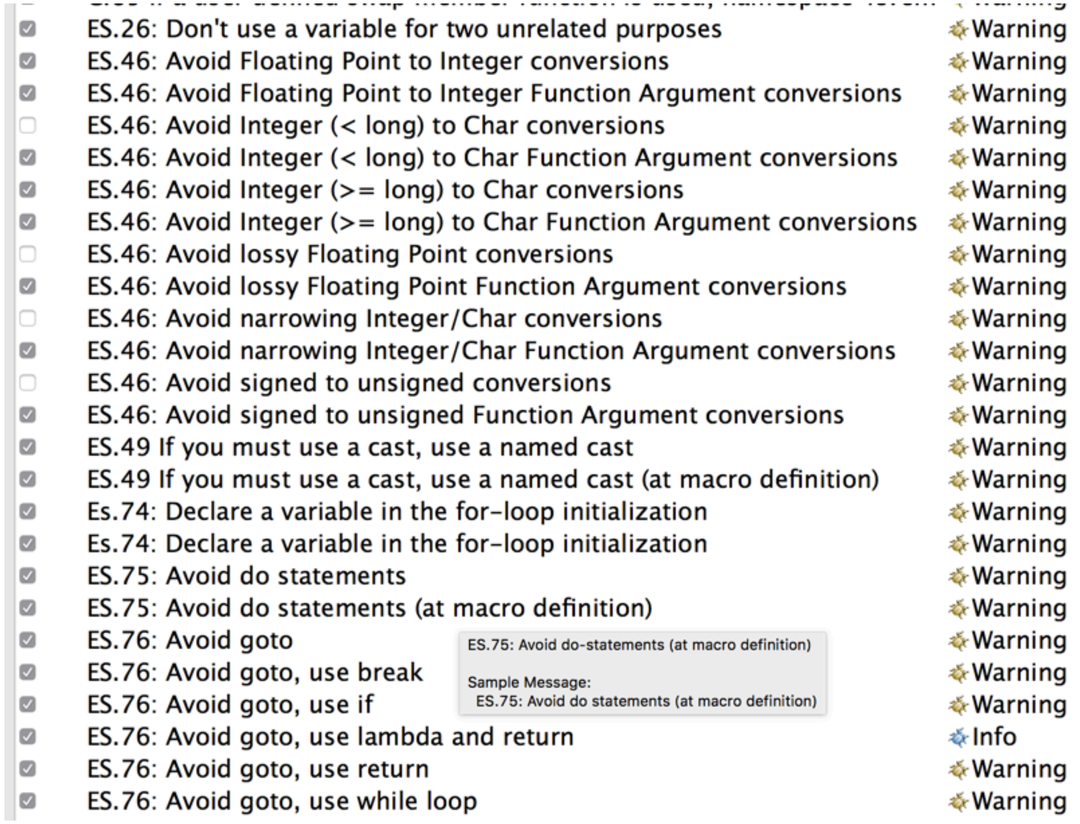
work in progress, not finished

- **Pointer:**
  - owner<T*>
  - not_null<T*>
- **Contracts (temporarily as macros): (contracts might become a standard feature)**
  - Expects(cond) - precondition(s)
  - Ensures(cond) - postcondition
- **Util (bad name :-)**
  - scope guard with the factory function finally(FUNC) - hopefully replaced by std:: mechanism in C++20
  - protection against narrowing errors: narrow<int>(0xffffU)
  - at(array,index) as free function with out of bounds guard (for std::array, plain arrays and containers with index op)
- **span/string_span (almost like string_view (17), but read/write instead of read only, will be in C++20)**
  - aka array_view (not std) and string_view (in C++17), but different and writable
  - goal: get rid of plain pointers representing arrays and strings (char*) - array decays to pointer as argument
  - string_span types for char, wchar_t, and const versions, not others (yet)

Support implemented in Cevelop and Core Guidelines Beta Plug-in



Name	Severity
Constificator Problems	
Missing const-qualification	Warning
Possibly missing const-qualification	Info
Cpp Core Guideline Problems	
C.164: Avoid conversion operators	Warning
C.20: Avoid redundant default operations	Warning
C.21: Missing special member functions	Warning
C.31: Destructor has no body	Warning
C.31: Destructor needed because of owner in member variables	Warning
C.31: Missing delete statements of owning member variables	Warning
C.35: A base class destructor should be either public and virtual, or pr...	Warning
C.37: Destructor should be declared noexcept	Warning
C.44: Constructor should be declared noexcept	Warning
C.45: Default constructor shouldn't only initialize data members	Warning
C.45: Default constructor shouldn't only initialize data members	Warning
C.46: Single-argument constructor should be declared explicit	Warning
C.47: Member variables should be initialized in the same order as they...	Warning
C.48: Prefer in-class initializers to member initializers in constructors...	Warning
C.49: Prefer initialization to assignment in constructors	Warning
C.60: Copy assignment should be non-virtual	Warning
C.60: Parameter should be taken by const&	Warning
C.60: Return parameter should be non-const&	Warning
C.63: Move assignment should be non-virtual	Warning
C.63: Return parameter should be non-const&	Warning
C.66: Move operations should be declared noexcept	Warning
C.83: Generate swap function	Warning
C.83: Swap Function Parameter has to be Reference	Warning
C.84 Make Swap noexcept	Warning
C.85 If a user defined swap member function is used, namespace-leve...	Warning



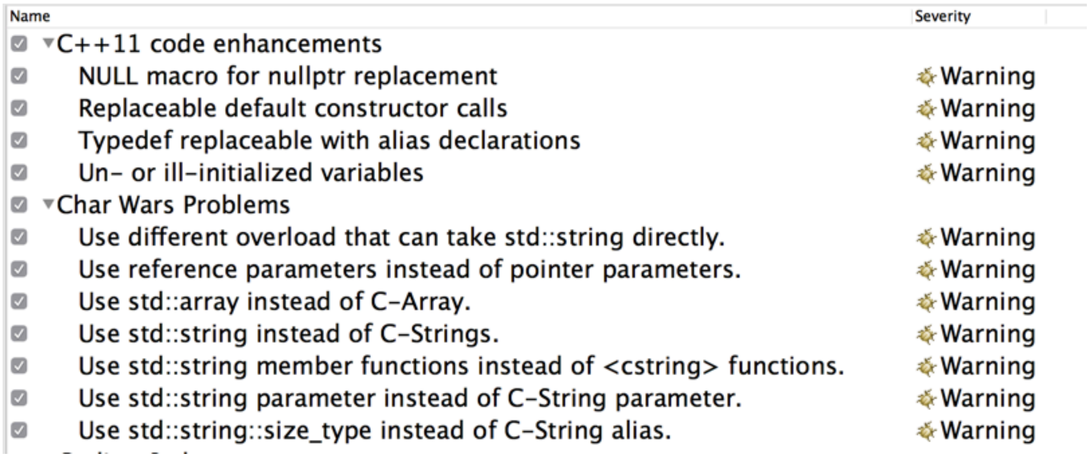
ES.26: Don't use a variable for two unrelated purposes	Warning
ES.46: Avoid Floating Point to Integer conversions	Warning
ES.46: Avoid Floating Point to Integer Function Argument conversions	Warning
ES.46: Avoid Integer (< long) to Char conversions	Warning
ES.46: Avoid Integer (< long) to Char Function Argument conversions	Warning
ES.46: Avoid Integer (>= long) to Char conversions	Warning
ES.46: Avoid Integer (>= long) to Char Function Argument conversions	Warning
ES.46: Avoid lossy Floating Point conversions	Warning
ES.46: Avoid lossy Floating Point Function Argument conversions	Warning
ES.46: Avoid narrowing Integer/Char conversions	Warning
ES.46: Avoid narrowing Integer/Char Function Argument conversions	Warning
ES.46: Avoid signed to unsigned conversions	Warning
ES.46: Avoid signed to unsigned Function Argument conversions	Warning
ES.49 If you must use a cast, use a named cast	Warning
ES.49 If you must use a cast, use a named cast (at macro definition)	Warning
Es.74: Declare a variable in the for-loop initialization	Warning
Es.74: Declare a variable in the for-loop initialization	Warning
ES.75: Avoid do statements	Warning
ES.75: Avoid do statements (at macro definition)	Warning
ES.76: Avoid goto	Warning
ES.76: Avoid goto, use break	Warning
ES.76: Avoid goto, use if	Warning
ES.76: Avoid goto, use lambda and return	Info
ES.76: Avoid goto, use return	Warning
ES.76: Avoid goto, use while loop	Warning

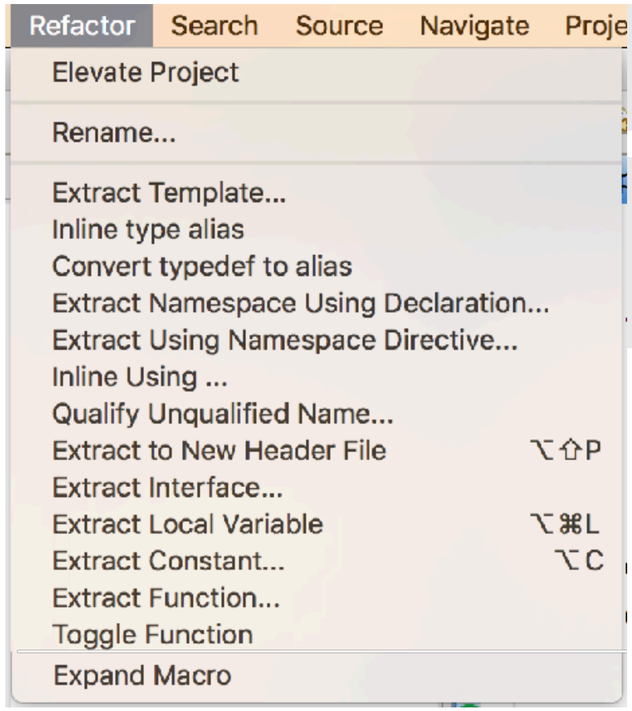
ES.75: Avoid do-statements (at macro definition)

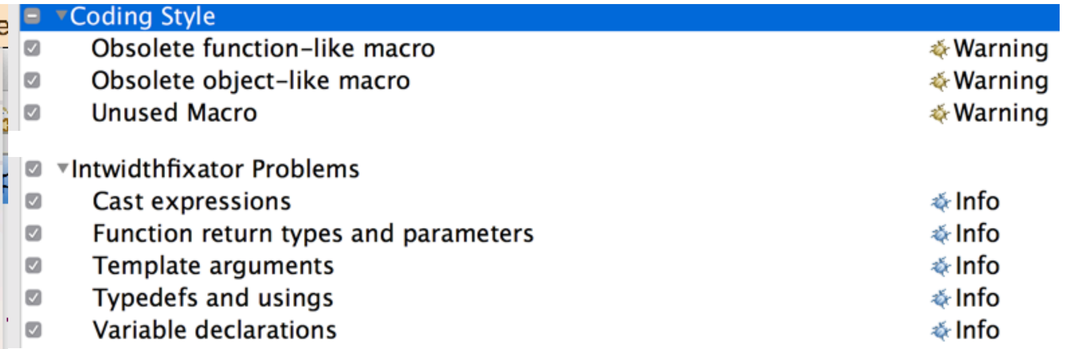
Sample Message:
ES.75: Avoid do statements (at macro definition)

Other support related to C++ Core Guidelines and modernizing Code bases

- **“Elevator” - C++11 updates for your code**
- **“CharWars” - substitute C-strings with std::string**
  - and plain arrays with std::array
- **“Macronator” - eliminate macros**
- **“IntWidthFixator” - Define fixed width integral types**
- **Refactorings**



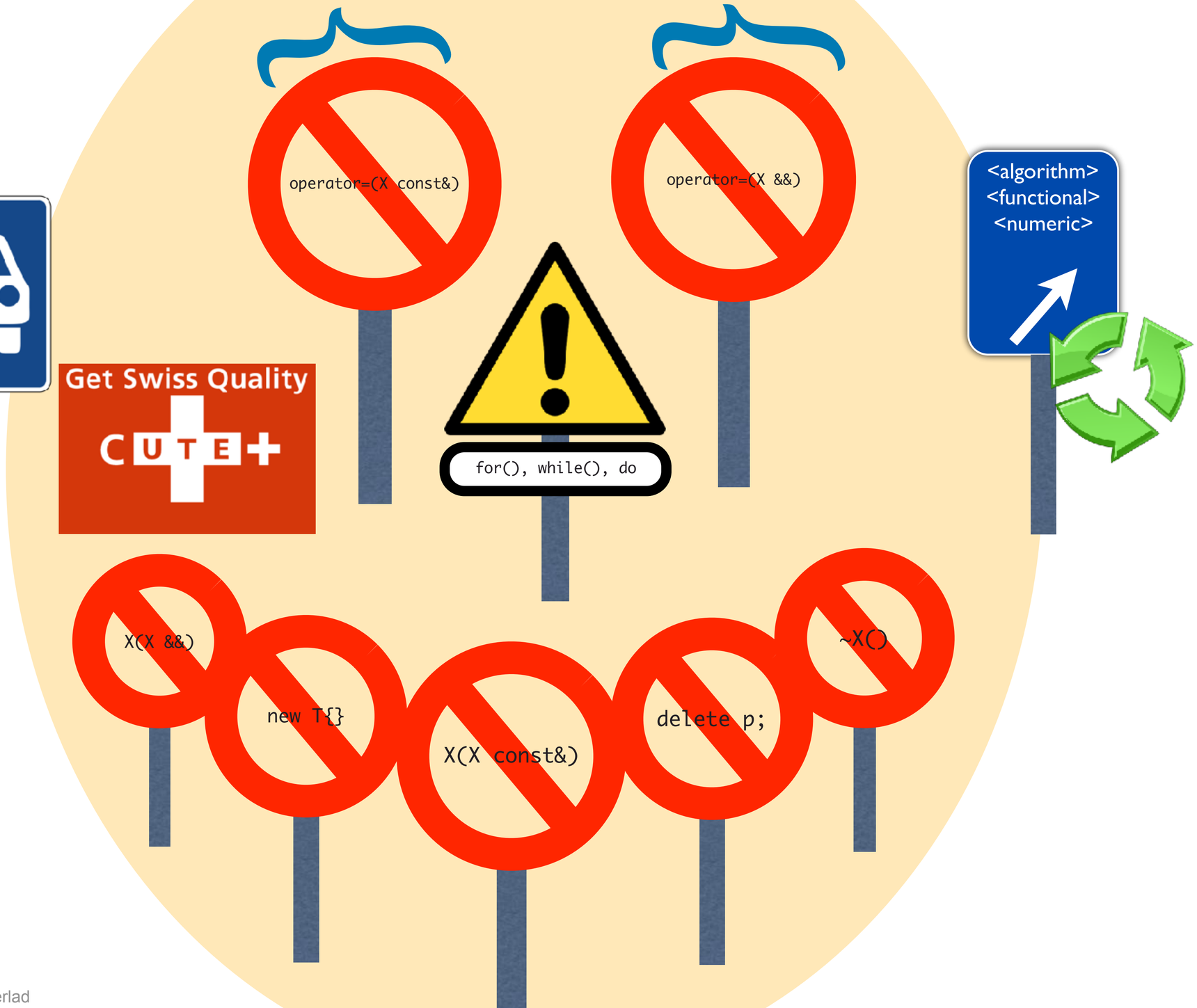




Wrap up

- **C++ Core guidelines can help migrating older C++ to more modern style**
  - but they are not done (and in some areas progress is slow after initial effort)
  - assume support with static analysis (Microsoft employs clang-analyze to achieve that in demos)
  - Cevelop tries to provide migration checkers, quick-fixes and refactorings -> you can help!
- **Take the guidelines and the GSL with a “grain of salt”**
  - some areas are still preliminary
  - some guidelines could be disputed
  - some curation and editing is required, today mostly just a collection
  - some rules are really old stuff (you already follow them, may be even unconsciously)
  - some GSL mechanism could be obsoleted by the C++ standard (but GSL is here today)
- **VS 2017 and Cevelop provide checkers (and some quick-fixes) for some of the guidelines**

Remember



Get Swiss Quality

CUTE+

Questions?

peter.sommerlad@hsr.ch
@PeterSommerlad

By the way, thanks for the great piece of software! This is by far the best free IDE for C/C++ so far after trying basically all the free C/C++ IDE. motowizlee on github 27.04.2017