The C++ Core Guidelines für sicheren Code

Rainer Grimm

Training, Coaching und Technologieberatung

www.ModernesCpp.de

Guidelines

Best Practices für die Verwendung von C++

- Warum benötigen wir Guidelines?
  - C++ ist eine anspruchsvolle Sprache in einer anspruchsvollen Domäne.
  - Alle drei Jahre erscheint ein neuer C++-Standard.
  - C++ wird in sicherheitskritischen Bereichen verwendet.

Hinterfrage deine Praktiken.

Bekanntesten Guidelines

- MISRA C++
  - **M**otor **I**ndustry **S**oftware **R**eliability **A**ssociation
  - Basierend auf MISRA C
  - Industriestandard im Automobil-, Flugzeug- und Medizinbereich
  - Veröffentlicht 2008  C++03

- AUTOSAR C++14
  - Basiert auf C++14
  - Wir zunehmend im Automobilbereich eingesetzt

- C++ Core Guidelines
  - Von der Community getragen

Überblick

- Philosophy
- Interfaces
- Functions
- Classes and class hierarchies
- Enumerations
- Resource management
- Expressions and statements
- Error handling
- Constants and immutability
- Templates and generic programming
- Concurrency
- The standard library
- Guideline support library

Aufbau

- Um die 350 Regeln und ein paar hundert Seiten.

- Jede Regel folgt einer ähnlichen Struktur.
  - The rule
  - A rule reference number
  - Reason(s)
  - Example(s)
  - Alternative(s)
  - Exception(s)
  - Enforcement
  - See also(s)
  - Note(s)
  - Discussion

Guidelines Support Library (GSL)

Ein kleine Bibliothek um die C++ Core Guidelines zu unterstützen.

- Implementierungen verfügbar für
  - Windows, Clang und GCC
  - GSL-lite für C++98 und C++03

- Komponenten
  - Views (keine Besitzer)
  - Owner (Besitzer)
  - Assertions (Kontrakte)
  - Utilities (Sichere Konvertierungen; verbesserte Threads)
  - Concepts

Interfaces

I.11: Never transfer ownership by a raw pointer (T*)

oder (Was lässt sich aus der Signatur einer Funktion ablesen):

- `func(value)`
  - `func` besitzt eine unabhängige Kopie des Wertes und die Laufzeit ist ihr Besitzer
- `func(pointer*)`
  - `pointer` ist ausgeliehen, kann aber ein Nullzeiger sein
  - `func` ist nicht der Besitzer und darf die Ressource nicht freigeben
- `func(reference&)`
  - `reference` ist ausgeliehen, besitzt aber immer einen Wert
  - `func` ist nicht der Besitzer und darf die Ressource nicht freigeben

Interfaces

- `func(std::unique_ptr)`
  - Aufruf

```
std::unique_ptr<int> uniq = std::make_unique<int>(2014);
func(std::move(uniq));
```

  - `std::unique_ptr` ist der Besitzer der Ressource

- `func(std::shared_ptr)`
  - Aufruf

```
std::shared_ptr<int> shar = std::make_shared<int>(2011);
func(shar);
```

  - `std::shared_ptr` ist ein zusätzlicher Besitzer der Ressource
  - `std::shared_ptr` verlängert die Lebenszeit der Ressource

Interfaces

I.13: Do not pass an array as a single pointer

- Was passiert, wenn `n` falsch ist?

```
void copy(const int* p, int* q, int n); // copy from [p:p+n] to [q:q+n]
void draw(double* p, int n);            // poor interface; poor code
```

- Verwende `span` aus der GSL

```
void copy(span<const int> r, span<int> r2);  // copy r to r2
void draw(span<int> p);

int a[100];
int b[100];
...
copy(a, b);

std::vector<int> vec;
...
draw(vec);
```

Functions

F.43: Never (directly or indirectly) return a pointer or a reference to a local object

```
int* f()
{
    int fx = 9;
    // ...
    return &fx;   // BAD
}

int& f()
{
    int x = 7;
    // ...
    return x;     // BAD
}
```

Class

C.2: Use class if the class has an invariant; use struct if the data members can vary independently

- Die Attribute können beliebige Werte annehmen

```
struct Pair {
    string name;
    int volume;
};
```

- Die Attribute besitzen Invarianten

```
class Date {
public:
    // validate that {yy, mm, dd} is a valid date and initialize
    Date(int yy, Month mm, char dd);
    // ...
private:
    int y;
    Month m;
    char d;     // day
};
```

Class

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

Enum

Enum.3: Prefer enum classes over “plain” enums

```
enum struct Color2: char{
  red= 126,
  blue, // 127
  green // 128 => ERROR
};
```

```
main.cpp:4:3: error: enumerator value '128' is outside the range of underlying type 'char'
   green // 128 => ERROR
   ^~~~~
```

- Konvertieren nicht heimlich zu `int`.
- Verschmutzen nicht den globalen Namensraum.
- Der Defaulttyp ist `int`, kann aber angepasst werden.

Resource Management

R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization)

- RAII-Idiom (Resource Acquisition Is Initialization)
  - Die Lebenszeit einer Ressource wird an eine lokale Variable gebunden.
  - Die Ressource wird im Konstruktor initialisiert und im Destruktor freigegeben.

- Verwendet von
  - Containern der Standard Template Library `std::string`
  - Smart Pointern
  - Locks
  - `std::jthread` (C++20)

Resource Management

```
class ResourceGuard{
  private:
    const std::string resource;
  public:
    ResourceGuard(const std::string& res):resource(res){
      std::cout << "Acquire the " << resource << "." <<  std::endl;
    }
    ~ResourceGuard(){
      std::cout << "Release the "<< resource << "." << std::endl;
    }
};

int main(){

  ResourceGuard resGuard1{"memoryBlock1"};

  {
    ResourceGuard resGuard2{"memoryBlock2"};
  }

  try{
      ResourceGuard resGuard3{"memoryBlock3"};
      throw std::bad_alloc();
  }
  catch (std::bad_alloc& e){
      std::cout << e.what();
  }

}
```

Expressions and Statements

ES.28: Use lambdas for complex initialization, especially of const variables

```
widget x;   // should be const, but:
for (auto i = 2; i <= N; ++i) {             // this could be some
    x += some_obj.do_something_with(i);  // arbitrarily long code
}                                           // needed to initialize x
// from here, x should be const, but we can't say so in code in this style
```

➡ `widget x` sollte `const` sein

```
const widget x = [&]{
    widget val;                                 // widget has a default constructor
    for (auto i = 2; i <= N; ++i) {             // this could be some
        val += some_obj.do_something_with(i);  // arbitrarily long code
    }                                           // needed to initialize x
    return val;
}();
```

Expressions and Statements

ES.100: Don’t mix signed and unsigned arithmetic

```cpp
#include <iostream>

int main(){

  int x = -3;
  int y = 7;

  std::cout << x - y << std::endl;  // -10
  std::cout << x + y << std::endl;  // 4
  std::cout << x * y << std::endl;  // -21
  std::cout << x / y << std::endl;  // 0

}
```

➡ *mixed arithmetic* mit GCC, Clang und MSVC

```cpp
#include <iostream>

int main(){

  int x = -3;
  unsigned int y = 7;

  std::cout << x - y << std::endl;  // 4294967286
  std::cout << x + y << std::endl;  // 4
  std::cout << x * y << std::endl;  // 4294967275
  std::cout << x / y << std::endl;  // 613566756

}
```

Concurrency and Parallelism

CP.8: Don’t try to use volatile for synchronization

- `std::atomic`
  - Atomarer (thread-safe) Zugriff auf geteilten Zustand.
- `volatile`
  - Zugriff auf speziellen Speicher, für den bestimmte Lese- und Schreiboptimierungen nicht erlaubt sind.

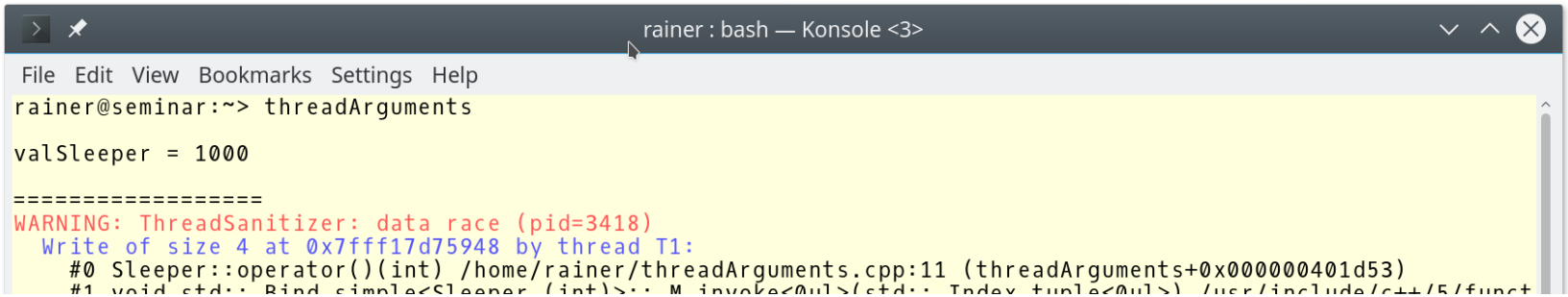
Java volatile == C++ atomic != C++ volatile

Concurrency and Parallelism

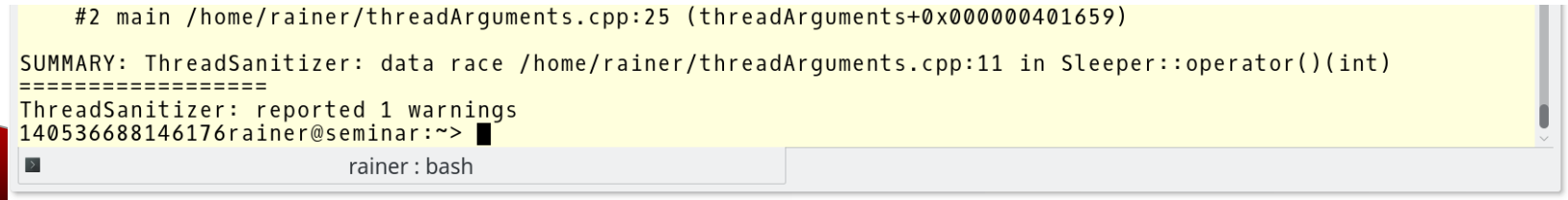
CP.9: Whenever feasible use tools to validate your concurrent code

Thread Sanitizer entdeckt Data Races zur Laufzeit.

```
g++ threadArguments.cpp -fsanitize=thread -g -o threadArguments
```



skip



Concurrency and Parallelism

CppMem: Interactive C/C++ memory model

(1)

Model

standard ◉ preferred ○ release_acquire ○ tot ○ relaxed_only

Program

examples/Paper | sc_atomics.c

◉ C ○ Execution

```
// contrasting with data_race.c, this
// shows a concurrent use of sc_atomic that does
// not have a data race
int main() {
  atomic_int x = 2;
  int y = 0;
  {{{ x.store(3);
  ||| y = ((x.load())==3);
  }}};
  return 0; }
```

(2)

run | reset | help **8 executions; 2 consistent, all race free**

Computed executions

(3)

Display Relations

☑sb ☐asw ☐dd ☐cd
☑rf ☑mo ☑sc ☑lo
☐hb ☐vse ☐ithb ☑sw ☐rs ☐hrs ☑dob ☐cad
☑unsequenced_races ☑data_races

(4)

Display Layout

○dot ○neato_par ◉neato_par_init ○neato_downwards

☐tex

edit display options

(5)

Execution candidate no. 1 of 8

previous consistent | previous candidate | next candidate | next consistent | 1 | goto

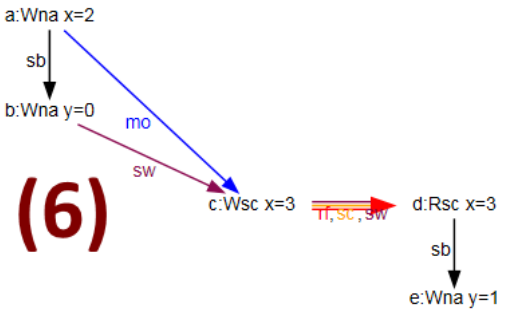
Model Predicates

consistent_race_free_execution = **true**

☑ consistent_execution = **true**
☑ assumptions = **true**
☑ well_formed_threads = **true**
☑ well_formed_rf = **true**
☑ locks_only_consistent_locks = **true**
☑ locks_only_consistent_lo = **true**
☑ consistent_mo = **true**
☑ sc_accesses_consistent_sc = **true**
☑ sc_fenced_sc_fences_heeded = **true**
☑ consistent_hb = **true**
☑ consistent_rf = **true**
☑ det_read = **true**
☑ consistent_non_atomic_rf = **true**
☑ consistent_atomic_rf = **true**
☑ coherent_memory_use = **true**
☑ rmw_atomicity = **true**
☑ sc_accesses_sc_reads_restricted = **true**

unsequenced_races are **absent**
data_races are **absent**
indeterminate_reads are **absent**
locks_only_bad_mutexes are **absent**

(6)



Files: out.exc, out.dot, out.dsp, out.tex

Error Handling

E.7: State your preconditions

E.8: State your postconditions

- **Precondition**: Soll beim Aufruf der Funktion gelten.
- **Postcondition**: Soll beim Beenden der Funktion gelten.
- **Assertion**: Soll an der Stelle gelten, an der die Zusicherung verwendet wurde.

```
int push(queue& q, int val)
  [[ expects: !q.full() ]]
  [[ ensures !q.empty() ]]{
  ...
  [[assert: q.is_ok() ]]
  ...
}
```

Constants and Immutability

Con.2: By default, make member functions const

```
struct Immutable{
    std::mutex m;
    int read() {
        std::lock_guard<std::mutex> lck(m);
        // critical section
        ...
    }
};
```

Die Methode `read` soll konstant sein!

Constants and Immutability

- **Physical constness**:
  - Das Objekt ist konstant und kann nicht verändert werden.
- **Logical constness**:
  - Das Objekt ist konstant und kann verändert werden.

```
struct Immutable{
    mutable std::mutex m;
    int read() const {
        std::lock_guard<std::mutex> lck(m);
        // critical section
        ...
    }
};
```

Templates and Generic Programming

T.10: Specify concepts for all template arguments

- **Concepts** sind Compile-Zeit Prädikate.

- Verwendung

```
template<Integral T>
T gcd(T a, T b){
  if( b == 0 ){ return a; }
  else{
    return gcd(b, a % b);
  }
}
```

- Definition

```
template<typename T>
concept bool Integral(){
  return std::is_integral<T>::value;
}
```

Templates and Generic Programming

- Core language concepts
  - Same
  - DerivedFrom
  - ConvertibleTo
  - Common
  - Integral
  - SignedIntegral
  - UnsignedIntegral
  - Assignable
  - Swappable
- Comparison concepts
  - Boolean
  - EqualityComparable
  - StrictTotallyOrdered
- Object concepts
  - Destructible
  - Constructible
  - DefaultConstructible
  - MoveConstructible
  - CopyConstructible
  - Movable
  - Copyable
  - Semiregular
  - Regular
- Callable concepts
  - Callable
  - RegularCallable
  - Predicate
  - Relation
  - StrictWeakOrder

Templates and Generic Programming

```
template <class T>
concept bool Integral() {
    return is_integral<T>::value;
}

template <class T>
concept bool SignedIntegral() {
    return Integral<T>() && is_signed<T>::value;
}

template <class T>
concept bool UnsignedIntegral() {
    return Integral<T>() && !SignedIntegral<T>();
}
```

Mehr zu den C++ Core Guidelines

- C++ Core Guidelines
- Artikel zu den C++ Core Guidelines
  - Englisch: www.ModernesCpp.com
    - Start Here
  - Deutsch: Modernes C++ auf heise Developer
    - Hier starten