Clean Code in C++

Rainer Grimm

Training, Coaching und Technologieberatung

Clean Code

Kontakt Person

- Rainer Grimm
- Telefon +49 7472 917441
- Mobil: +49 1523 1965939
- E-Mail: schulung@ModernesCpp.de

Verlauf

Version	Datum	Beschreibung	Author
1.0	2020-03-24	Initiale Version	Grimm Rainer
1.1	2020-10-08	Überarbeitung des Layouts	Grimm Juliette Grimm Rainer

Clean Code

Clean Code is code that is easy to understood und easy to change (Cvuorinen.net, Carl Vuorinen).

- Leicht zu verstehen
  - Ausführungsablauf der Anwendung
  - Wechselwirkung der Objekte
  - Zweck der Funktionen
  - Zweck der Ausdrücke und Variablen
- Leicht zu ändern
  - kleine Klassen und Funktionen mit einer einzigen Verantwortung
  - klare und übersichtliche Interfaces
  - Code ist leicht testbar und verfügt über Unit-Tests
  - Tests sind leicht zu verstehen und zu verändern

Clean Code

- **Elegant**: Clean Code ist angenehm zu lesen, sollte Sie zum Lächeln bringen.
- **Lesbarkeit:** Clean Code sollte sich wie gut geschriebene Prosa lesen lassen.
- **Einfach:** Machen Sie eine Sache mit dem Single Responsibility Prinzip (SRP).
- **Testbar:** Führen Sie alle Tests durch.

(Hackernoon, Luan Nguyen)

Clean Code in C++

- Wichtige Grundsätze
- Best Practices
- Pattern und Idiome
- Testen
- Refactoring
- Werkzeugunterstützung

Clean Code in C++

- Wichtige Grundsätze
- Best Practices
- Pattern und Idiome
- Testen
- Refactoring
- Werkzeugunterstützung

KISS

Keep **I**t **S**imple, **S**tupid or **K**eep **I**t **S**imple and **S**tupid.

- Tun Sie die einfachste Sache, die überhaupt funktionieren könnte.

- Grund
  - Einfachheit sollte das Hauptziel der Software-Entwicklung sein.
  - Unnötige Komplexität sollte vermieden werden.
  - Software hat bereits eine implizite Komplexität.
  - Fügen Sie nur das Maß an Komplexität hinzu, das die Sache einfacher gestaltet.

YAGNI

You **A**rent't **G**onna **N**eed **I**t or **Y**ou **A**in't **G**onna **N**eed **I**t.

- Grund
  - stark verwandt mit KISS
  - gegen spekulative Verallgemeinerung und Over-Engineering
  - Produzieren Sie nicht etwas für mögliche, zukünftige Use-Cases.
  - Der unnötige Code ist verschwendete Zeit, verkompliziert den Code und führt zu Bugs.
  - Refaktorieren Sie ihren Code, falls erforderlich.

DRY

Don’t **R**epeat **Y**ourself or Once und Only Once (OOAO).

- Grund
  - Eine der am häufigsten (unbeabsichtigt oder absichtlich) verletzten Regeln.
  - Wenn sich ein Stück des Codes ändert, dann vielleicht nicht das andere.
  - *"every piece of knowledge must have a single, unambiguous, authoritative representation within the system*" (*The Pragmatic Programmer*)

PLA

Principle of **L**east **A**stonishment or **P**rinciple of **L**east **S**urprise.

- Grund
  - Der Nutzer sollte nicht von einem unerwarteten Verhalten oder mysteriösen Nebeneffekten des User-Interface überrascht sein.
  - Eine Funktion sollte genau das tun, was der Funktionsname impliziert.
  - "*Make interfaces easy to use correctly und hard to use incorrectly*. " (Scott Meyers)

Boy Scout Rule

Always leave the campground cleaner than you found it.

- Grund
  - Wann immer Sie ein Stück Code finden, welches verbessert werden sollte, überarbeiten Sie dies.
  - Code wird mit dieser Regel nicht verfallen.
  - Kleine Schritte verbessern den Code.
    - Umbenennen schlechter Namen
    - Dekomposition großer Funktionen
    - Löschen des Kommentars, indem er selbsterklärend gemacht wird
    - komplexen Code aufräumen
    - Entfernen von Duplikaten

SOLID

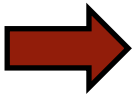
- Single responsibility principle
  - Eine Software-Einheit sollte ein Zuständigkeit haben.
- Open–closed principle
  - Software-Einheiten sollten offen für Erweiterungen, aber geschlossen für Änderungen sein.
- Liskov substitution principle
  - Objekte in einem Programm sollten durch Instanzen ihrer Untertypen ersetzbar sein, ohne die Korrektheit des Programms zu verändern.
- Interface segregation principle
  - Viele kundenspezifische Interfaces sind besser als ein universelles Interface.
- Dependency inversion principle
  - Man sollte sich auf die Interfaces verlassen, nicht auf die Implementierung.

Clean Code in C++

- Wichtige Grundsätze
- **Best Practices**
- Pattern und Idiome
- Testen
- Refactoring
- Werkzeugunterstützung

C++ Core Guidelines

- Warum brauchen wir Best Practices?
  - C++ ist eine komplexe Sprache in einer komplexen Umgebung.
  - Alle drei Jahre wird ein neuer C++-Standard veröffentlicht.
  - C++ wird in sicherheitskritischen Systemen eingesetzt.



C++ Core Guidelines besitzt von der Community erstellte Richtlinien zum Schreiben guter Software.

C++ Core Guidelines

Die C++ Core Guidelines bestehen aus 350 Regeln und 600 Seiten.

- Sections
  - Philosophie
  - Interfaces
  - Funktionen
  - Klassen und Klassenhierachien
  - Aufzählungen
  - Ressource Management
  - Expressions und Statements
  - Performanz
  - Concurrency
  - Error Handling
  - Konstanten und Immutabilität
  - Templates und generische Programmierung
  - C-Style Programmierung
  - Die Standardbibliothek
- Unterstützende Abschnitte
  - Guidelines Support Library

Philosophie

Die philosophischen Regeln liefern Begründungen für die folgenden konkreten Regeln. Im Idealfall lassen sich die konkreten Regeln aus den philosophischen Regeln ableiten.

- Absicht und Ideen direkt in Code ausdrücken.
- Schreiben Sie in ISO Standard C++ und verwenden Sie Bibliotheken und Werkzeuge.
- Ein Programm sollte statisch typsicher sein und daher zur Compiletime überprüft werden. Wenn dies nicht möglich ist, fangen Sie Laufzeitfehler frühzeitig ab.
- Verschwenden Sie keine Ressourcen wie Platz oder Zeit.
- Kapseln Sie unsaubere Konstrukte hinter einer stabilen Schnittstelle.

Philosophie

- Übungen:

  - Verschaffen Sie sich einen Überblick zu den Regeln der C++ Core Guidelines.

- Weitere Informationen:
  - Philosophie

Interfaces

Eine Schnittstelle ist ein Vertrag zwischen einem Service Anbieter und einem Service Nutzer.

Vermeiden Sie globale Variablen und Singletons. Sie brechen

- Testbarkeit
- Refaktoring
- Optimierung
- Concurrency

```
int glob{2011};
int multiply(int fac) {
    glob *= glob;
    return glob * fac;
}
```

singleton.cpp
singletonMeyer.cpp

Interfaces

Die Schnittstellen sollten

- explizit sein.
- präzise und stark typisiert sein.
- eine geringe Anzahl von Argumenten haben.
- ähnliche Argumente trennen.

```
void showRectangle(double a, double b, double c, double d) {
    a = floor(a);
    b = ceil(b);
    ...
}

void showRectangle(Point top_left, Point bottom_right);
```

Interfaces

Übergeben Sie keine Arrays mittels Zeiger.

```
template <typename T>
void copy_n(const T* p, T* q, int n) { ... }

template <typename T>
void copy(std::span<const T> src, std::span<T> des){ ... }

...

int a[100] = {0, };
int b[100] = {0, };
copy_n(a, b, 101);
copy(a, b);
```

Interfaces

- Beispiele:
  - `singleton.cpp`
  - `singletonMeyer.cpp`
- Übungen:
  - Schnittstellen haben einen funktionalen und einen nicht-funktionalen Datenkanal. Was ist der funktionale und der nicht-funktionale Datenkanal?
  - Funktionen unterstützen per Design Interfaces. Welche anderen Softwarekomponenten sollten ebenfalls Interfaces haben?
  - Das Singleton-Pattern wird teilweise als Pattern und teilweise als Anti-Pattern gesehen. Was sind die Vor- und Nachteile des Singleton-Pattern?
- Weitere Informationen:
  - Interfaces

Funktionen

Software-Entwickler beherrschen Komplexität, indem sie komplexe Aufgaben in kleinere Einheiten aufteilen. Funktionen sind "*the most critical part in most interfaces*".

- Funktionen sollten eine Operation durchführen. Die Vorteile sind
  - gute Namen per Design.
  - kurze Funktionen, die leicht verständlich sind.
  - Testbarkeit nach Konstruktion.

Funktionen

Machen Sie Ihre Funktionen `constexpr` wenn möglich.

- `constexpr` Funktionen
  - haben das Potenzial, zur Compiletime ausgeführt zu werden.
  - sind fast rein.
  - sind thread-sicher, wenn sie zur Compiletime ausgeführt werden.

```
constexpr auto gcd(int a, int b) {
    while (b != 0) {
        auto t= b;
        b = a % b;
        a = t;
    }
    return a;
}
```

constexpr.cpp

Funktionen

Unterscheidung zwischen in, in/out und out Parameter

<table>
<tr><th></th><th>Billig oder unmöglich zu kopieren</th><th>Niedrige oder moderate Kosten zu verschieben oder unbekannte Kosten</th><th>Teuer zu verschieben</th></tr>
<tr><td>In</td><td rowspan="2"><code>func(X)</code></td><td colspan="2" rowspan="2"><code>func(const X&)</code></td></tr>
<tr><td>In & erhalte “copy”</td></tr>
<tr><td>In & move from</td><td colspan="3"><code>func(X&&)</code></td></tr>
<tr><td>In/Out</td><td colspan="3"><code>func(X&)</code></td></tr>
<tr><td>Out</td><td colspan="2"><code>X func()</code></td><td><code>func(X&)</code></td></tr>
</table>

- Billig oder unmöglich zu kopieren: `int` or `std::unique_ptr`
- Billig zu verschieben: `std::vector` or `std::string`
- Moderate Kosten zu verschieben: `std::vector<BigPOD>`
- Unbekannt: `template`
- Teuer zu verschieben: `std::array<BigPod>`

Funktionen

Kosten der Operationen:

- günstig zu kopieren: <= 3 ints
- billig oder moderate Kosten zu verschieben: <= 1000 Bytes ohne Speicherallokation
- aufwendig zu verschieben: >= 1000 Bytes

RVO (Return Value Optimization)

```
Type f() {
  return Type{}; // no copy
}
Type my = f();   // no copy
```

NRVO (Named Return Value Optimization)

```
Type f() {
  Type myVal;
  return myVal; // one copy
}
Type my = f();  // no copy
```

Funktionen

Besitzverhältnisse von Funktionparametern.

Beispiel	Besitzsemantik
`func(value)`	`func` ist ein unabhängiger Besitzer der Ressource
`func(pointer*)`	`func` hat die Ressource geliehen
`func(reference&)`	`func` hat die Ressource geliehen
`func(std::unique_ptr)`	`func` ist ein unabhängiger Besitzer der Ressource
`func(std::shared_ptr)`	`func` ist ein Mitbesitzer der Ressource

`ownershipSemantic.cpp`

Funktionen

- Beispiele:

  - `constexpr.cpp`
  - `ownershipSemantic.cpp`

- Übungen:
  - Die Funktion `read_und_print` hat viele Probleme:

```
void read_und_print() {
    int x;
    std::cin >> x; // check für errors
    std::cout << x << '\n';
}
```

    - Refaktorien Sie die Funktion und benutzen Sie sie.
    - Lösung: `readundPrint.cpp`

Funktionen

- Übungen:

  - Analysieren Sie das Programm `constexpr.cpp` mit der Hilfe des Compiler Explorers.
    - Welche sind die Vorteile der `constexpr` Funktionen?

- Weitere Informationen:

  - Funktionen

Klassen und Klassenhierachien

Eine Klasse ist ein benutzerdefinierter Typ, bei dem der Programmierer das Verhalten spezifiziert.
Klassenhierachien organisieren verwandte Klassen zu hierarchischen Strukturen.

`class` versus `struct`

- Nutzen Sie eine Klasse, wenn sie eine Invariante hat.
- Setzen Sie die Invariante in dem Konstruktor.

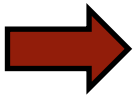
```
struct Point {
    int x;
    int y;
};
```

```
class Date {
 public:
     Date(int yy, Month mm, char dd);
 private:
     int y;
     Month m;
     char d;
};
```

Konkrete Datentypen

Ein konkreter Datentyp (value typ) ist kein Teil einer Typhierachie. Es kann auf der Stack erzeugt werden.

Ein konkreter Typ sollte regulär sein.

- Default Konstruktor: `X()`
- Copy Konstruktor: `X(const X&)`
- Copy Zuweisung: `operator = (const X&)`  **Big Six**
- Move Konstruktor: `X(X&&)`
- Move Zuweisung: `operator = (X&&)`
- Destruktor: `~(X)`
- Swap Operator: `swap(X&, X&)`
- Gleichheitsoperator: `operator == (const X&)`

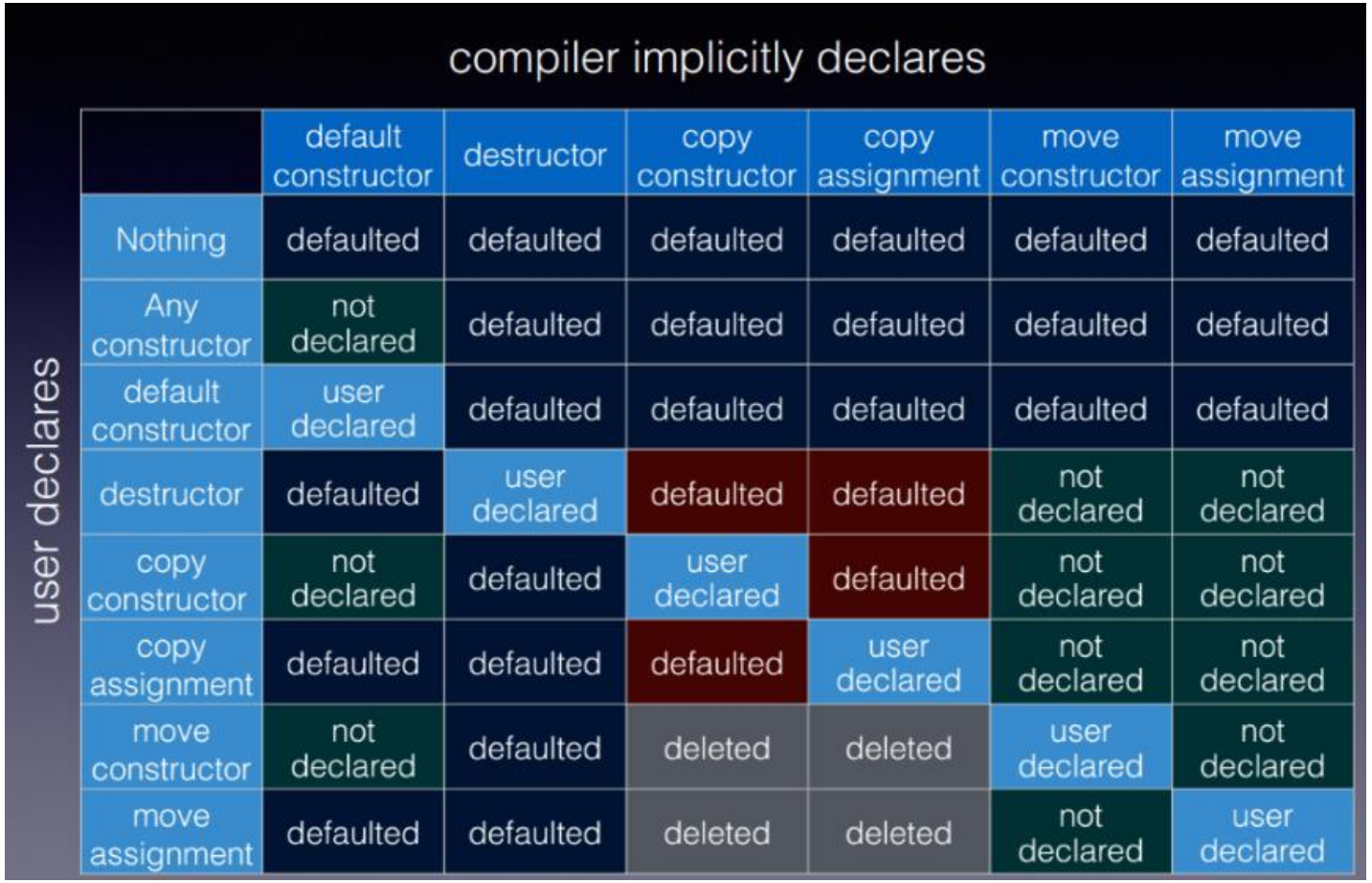
Klassen und Klassenhierarchien

Die Big Six

- Der Compiler kann sie automatisch erzeugen.
- Sie können eine spezielle Memberfunktion per `default` anfordern.
- Sie können eine generierte Funktion per `delete` löschen.
- Definieren Sie alle oder keinen von ihnen (Sechserregel oder Nullregel).
- Definieren Sie sie konsistent.
- Es gibt Abhängigkeiten zwischen den Big Six.

`delete.cpp`
`swap.cpp`

Klassen und Klassenhierarchien



by Howard Hinnant

- user declared: eine Methode, die verwendet wird (definiert, `default`ed oder `delete`d)
- defaulted: eine Methode, die der Compiler generiert oder standardmäßig angefordert wird

Klassen und Klassenhierarchien

- Beispiele:

  - `delete.cpp`
  - `swap.cpp`
  - `bigArray.cpp`

- Übungen:
  - In dem Programm `bigArray.cpp` wird ein BigArray mit 10 Milliarden Einträgen auf einen `std::vector` geschoben.
  - Übersetzen Sie das Programm und messen Sie seine Performanz.
  - Erweitern Sie das `BigArray` um Move Semantik und messen Sie die Performanz noch einmal. Wie groß ist der Gewinn der Performanz?
    - Lösung: `bigArray.cpp`

Klassen und Klassenhierarchien

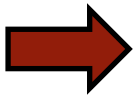
- Weitere Informationen:

  - Klassen und Klassenhierachien
  - Rvalue References Explained von Thomas Becker

Konstruktor

Definieren Sie keinen Standard-Konstruktor, der nur Datenelemente initialisiert; verwenden Sie stattdessen die Elementinitialisierung

```
struct Widget {
    Widget() = default;
    Widget(int w): width(w) {}
 private:
     int width = 640;
};
```



Definieren Sie das Standardverhalten der einzelnen Objekte im Klassenkörper. Verwenden Sie explizite Konstruktoren für Variationen des Standardverhaltens.

classMemberInitializerWidget.cpp

Konvertierungs-Konstruktor und -Operator

Definieren Sie Konstruktoren, die ein Argument erhalten (Konvertierungs-Konstruktor) und Konvertierungs-Operatoren `explicit`.


```
class MyClass{
  public:
    explicit MyClass(A){}     // converting constructor
    explicit operator B(){}   // converting operator
};
```

conversionOperator.cpp

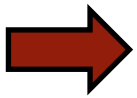
Delegating Konstruktor

Verwenden Sie den delegierenden Konstruktor, um gemeinsame Aktionen für alle Konstruktoren einer Klasse anzubieten.

```
class Degree {
 public:
    explicit Degree(int deg) {
        degree= deg % 360;
        if (degree < 0) degree += 360;
    }
    Degree(): Degree(0) {}
    explicit Degree(double deg):
        Degree(static_cast<int>(std::ceil(deg))) {}
 private:
    int degree;
};
```

Polymorphe Klassen

Eine polymorphe Klassen sollte das Kopieren unterdrücken.
Eine polymorphe Klasse ist eine Klasse, die mindestens eine virtuelle Funktion definiert oder erbt.

 Gefahr von Slicing

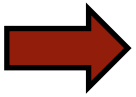
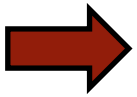
Slicing: Das Kopieren eines Objekts während der Zuweisung oder Initialisierung gibt nur einen Teil des Objekts zurück.

Destruktoren

- Definieren Sie einen Destruktor, wenn eine Klasse eine explizite Aktion bei der Objektzerstörung benötigt.
- Ein Destruktor der Basisklasse sollte entweder `public` und virtuell oder `protected` und nicht-virtuell sein.
  - `public` und virtuell:
    - Instanzen von abgeleiteten Klassen können durch einen Basisklassenzeiger zerstört werden; dasselbe gilt für Referenzen
  - `protected` und nicht-virtuell:
    - Instanzen abgeleiteter Klassen können nicht durch einen Basisklassenzeiger zerstört werden; dasselbe gilt für Referenzen
- Destruktoren sollten nicht fehlschlagen; machen Sie sie `noexcept`

Konstruktor/Destruktor (virtual)

Rufen Sie keine virtuellen Funktionen in Konstruktoren und Destruktoren auf.

- Rein virtuell:  undefined behavior
- Virtuell:  Virtualität ist deaktiviert

`virtualCall.cpp`

Operator Overloading

- Benutzen Sie nicht-member Funktionen für symmetrische Operatoren.

```
class MyInt {
    int num;
 public:
    explicit MyInt(int n): num(n) {};
    friend bool operator==(const MyInt& lhs, const MyInt& rhs) {
         return lhs.num == rhs.num;
    }
    friend bool operator==(int lhs, const MyInt& rhs) {
        return lhs == rhs.num;
    }
    friend bool operator==(const MyInt& lhs, int rhs) {
        return lhs.num == rhs;
    }
};
```

Klassen und Klassenhierarchien

- Beispiele:

  - `classMemberInitializerWidget.cpp`
  - `conversionOperator.cpp`
  - `convertingConstructor.cpp`
  - `slice.cpp`
  - `virtualCall.cpp`

Klassen und Klassenhierarchien

- Übungen:

  - Das Programm `convertingConstructor.cpp` unterstützt type-sichere Arithmetik mit benutzerdefinierten Literalen.
    - Führen Sie das Programm aus.
    - In dem Programm ist ein Fehler aufgetreten. Konstruktoren mit einem Argument sollten `explicit` sein. Beheben Sie diesen.
    - Erweitern Sie das Programm so, dass Fließkommazahlen zu den benutzerdefinierten Literalen hinzugefügt werden können. Lösung: `convertingKonstruktor.cpp`
  - Refaktorieren Sie die Konstruktoren.
    - Den Konstruktor der Klasse `Widget` in `classMemberInitializerWidget.cpp` kann vereinfacht werden. Verwenden Sie die direkte Initialisierung für die Klassenmitglieder.
      - Lösung: `classMemberInitializerWidget.cpp`

Klassen und Klassenhierarchien

- Weitere Informationen:

  - Klassen und Klassenhierarchien
  - Rvalue References Explained von Thomas Becker

Klassenhierarchien

Eine Klassenhierarchie repräsentiert eine Reihe von hierarchisch organisierten Konzepten. Basisklassen fungieren typischerweise als Schnittstellen.

- **Interface Vererbung** verwendet öffentliche Vererbung. Sie trennt Benutzer von der Implementierung, damit abgeleitete Klassen hinzugefügt und geändert werden können, ohne die Benutzer der Basisklasse zu beeinträchtigen.
- **Implementation Vererbung** verwendet oft private Vererbung. Typischerweise stellt die abgeleitete Klasse ihre Funktionalität zur Verfügung, indem sie die Funktionalität der Basisklassen anpasst.

Klassenhierarchien

Verwenden Sie Klassenhierarchien, um Konzepte mit inhärenter hierarchischer Struktur darzustellen.

```
template<typename T>
class Container {
 public:
       // list operations:
    virtual T& get() = 0;
    virtual void put(T&) = 0;
    virtual void insert(Position) = 0;
      // vector operations:
    virtual T& operator[](int) = 0;
    virtual void sort() = 0;
      // tree operations:
    virtual void balance() = 0;
};
```

Abstrakte Klassen

- Wenn eine Klasse als Interface verwendet wird, machen Sie sie zu einer abstrakten Klasse.
- Verwenden Sie abstrakte Klassen, wenn eine vollständige Trennung von Schnittstelle und Implementierung erforderlich ist.
- Eine abstrakte Klasse benötigt normalerweise keinen Konstruktor.

Virtualität

- Eine Klasse mit einer virtuellen Funktion sollte einen `public` und virtuellen oder einen `protected` Destruktor haben.

- Virtuelle Funktionen sollten genau eines der folgenen Schlüsselworte verwenden: `virtual`, `override` oder `final`.

Virtualität

- Für das Erzeugen tiefer Kopien von polymorphen Klassen sollen Sie einen virtuellen `clone` Funktion anstelle eines Copy-Konstruktor oder Copyzuweisungsoperator verwenden.

  ➡ Gefahr von slicen

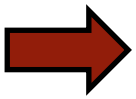
- **Kovarianter Rückgabetyp:** erlaubt es für eine überschreibende Member-Funktion, einen Subtyp des Rückgabetyps der überschreibenden Member-Funktion zurückzugeben.

`cloneFunction.cpp`

Fallen (Shadowing)

- Erstellen Sie eine Menge überladener Methoden für eine abgeleitete Klasse und ihre Basisklassen mit `using`.

```
struct Base {
    void func(double d) { std::cout << "f(double) \n"; }
};

struct Derived: public Base {
     void func(int i) { std::cout << "f(int) \n"; }
};
...
Derived der;
der.func(2011);    // f.int()
der.func(2020.5);  // f.int()
```



```
struct Derived: public Base {
    void func(int i) { std::cout << "f(int) \n"; }
    using Base::func; // exposes func(double)
};
```

Fallen (Virtuelle Funktionen und Defaults)

- Geben Sie keine Default-Argumente für eine virtuelle Funktion und eine überschreibende Methoden an.

```
struct Base {
    virtual int multiply(int value, int factor = 2) = 0;
};
struct Derived : public Base {
    int multiply(int value, int factor = 10) override {
        return factor * value;
    }
};
```

Unions

- Verwenden Sie `unions` um Speicherplatz zu sparen.
- Vermeiden sie "*naked*" `unions`.
- Verwenden Sie tagged `unions` (`std::variant`)

```
std::variant<int, float> v, w;
v = 12;
int i = std::get<int>(v);

w = std::get<int>(v);
w = std::get<0>(v);
w = v;
```

variant.cpp

Klassen und Klassenhierarchien

- Beispiele:

  - `cloneFunction.cpp`
  - `variant.cpp`

- Übungen:
  - C++ unterstützt Interface-Vererbung und Implementierungs-Vererbung.
    - Kennen Sie einen Anwendungsfall für Implementierungs-Vererbung?
    - Studieren Sie das Programm `adapter.cpp`, dass das adapter pattern mittels Mehrfachvererbung umsetzt.
    - Kennen Sie einen anderen Weg, das Adapter-Pattern zu implementieren?

Klassen und Klassenhierarchien

- Weisen Sie niemals einen Zeiger auf ein Array von abgeleiteten Klassenobjekten einem Zeiger auf seine Basisklasse zu.
- Welchen Fehler besitzt das Codebeispiel?

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived d[] = {{1, 2}, {3, 4}, {5, 6}};
Base* pB = d;
pB[1].x = 7;
```

- Weitere Informationen:
  - Klassen und Klassenhierarchien

Aufzählungen

Aufzählungen werden verwendet, um Mengen ganzzahliger Konstanten und auch einen Typ für solche Mengen von Werten zu definieren.

- Bevorzugen Sie `enum` Klassen gegenüber "einfachen" Aufzählungen.
- Geben Sie Aufzähler nur bei Bedarf an.

```
enum class Month: char {
  jan = 1,
  feb,
  ...
};
```

stronglyTypedEnum.cpp

Aufzählungen

- Beispiele:
  - `stronglyTypedEnum.cpp`

- Übungen:
  - Haben Sie Aufzähler für die integrale Arithmetik verwendet?

- Weitere Informationen:
  - Aufzählungen

Ressource Management

Eine Ressource ist etwas, das Sie verwalten müssen. Das heißt, Sie müssen sie anfordern und freigeben oder sie schützen.

- Die entscheidende Frage an eine Ressourcen ist: Wer ist der Besitzer?

Ressource Management: Besitz

- **Lokale Objekte:** Die C++-Laufzeit als Besitzer verwaltet automatisch die Ressourcen. Dies gilt für globale Objekte oder Klassenmitglieder.
- **Referenz**: Ich bin nicht der Besitzer. Ich habe nur die Ressource ausgeliehen, die nicht leer sein kann. Ich darf die Ressource nicht löschen.
- **Zeiger**: Ich bin nicht der Besitzer. Ich habe nur die Ressource ausgeliehen, die leer sein kann. Ich darf die Ressource nicht löschen.
- **`std::unique_ptr`**: Ich bin der alleinige Besitzer der Ressource.
- **`std::shared_ptr`**: Ich darf die Ressource nicht löschen: Ich teile die Ressource mit anderen `std::shared_ptr`. Ich darf den geteilten Besitz explizit freigeben.
- **`std::weak_ptr`**: Ich bin nicht der Besitzer der Ressource, aber ich kann der vorübergehende Besitzer der Ressource werden.

Ressource Management: RAII

RAII steht für **R**esource **A**cquisition **I**s **I**nitialization.

- Zentrale Idee:
  - Erstellen Sie ein lokales Proxy-Objekt für Ihre Ressource.
  - Der Konstruktor des Proxy-Objektes fordert die Ressource an und der Destruktor des Proxy-Objektes gibt die Ressource frei.
  - Die C++-Laufzeiten verwalten die Lebensdauer des Proxys und damit die der Ressource.
- Umsetzungen
  - Smart pointer
  - Locks
  - Container der STL
  - `std::jthread`

Ressource Management: NNN

NNN steht für **N**o **N**aked **N**ew und bedeutet, dass die Speicherzuweisung nicht als eigenständige Operation durchgeführt werden sollte, sondern in einem Managerobjekt.

Smart Pointer:

- **std::unique_ptr**: exklusiver Besitzer
- **std::shared_ptr**: geteilter Besitzer
- **std::weak_ptr**: kann Besitzer erzeugen

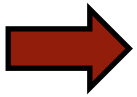
std::unique_ptr

- Allokiere die Ressource nicht außerhalb

```
int* myInt = new int(2011);
std::unique_ptr<int> uniq1 = std::unique_ptr<int>(myInt);
std::unique_ptr<int> uniq2 = std::unique_ptr<int>(myInt);
```

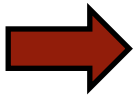
- Bevorzugen Sie `std::make_unique` gegenüber `std::unique_ptr`

```
func(std::unique_ptr<int>(new int(2011)),
     std::unique_ptr<int>(new int(2014)));
```



Möglicherweise Speicherleck (bis C++17)

```
func(std::make_unique<int>(2011),
     std::make_unique<int>(2014));
```



garantiert kein Speicherleck (Verbesserung der Performanz)

Bevorzugen Sie `std::unique_ptr` gegenüber `std::shared_ptr`.

std::shared_ptr

- Benutzen Sie std::shared_ptr nur für gemeinsame Besitzansprüche.
  - `std::unique_ptr` kann verschoben werden

```
void func(std::unique_ptr<int> myUniq);
…
auto myUniq = std::make_unique<int>(2014));
func(std::move(myUniq));
```

- Bevorzugen Sie `std::make_shared` gegenüber `std::shared_ptr`.
  - erzeugt keine Ausnahme
  - benötigt eine statt zwei Allokationen
- Der Kontrollblock ist thread-sicher, aber nicht die Ressource.
  - `std::atomic_shared_ptr` mit C++20
- Kann mit einem eigenen Deleter verwendet werden (entsprechend `std::unique_ptr`).

```
std::shared_ptr<int>(2011, Deleter());
```

sharedPtrDeleter.cpp

Smart Pointer als Parameter

Funktionsignatur	Bedeutung
`func(std::unique_ptr<int>)`	`func` ist ein exklusiver Besitzer
`func(std::unique_ptr<int>&)`	`func` kann `int` neu setzen
`func(std::shared_ptr<int>)`	`func` ist ein geteilter Besitzer
`func(std::shared_ptr<int>&)`	`func` kann `int` neu setzen
`func(const std::shared_ptr<int>&)`	`func` speichert den Referenzzähler

`func(const std::shared_ptr<int>&)` besitzt keinen Mehrwert gegenüber einem Zeiger oder einer Referenz.

Ressource Management

- Beispiele:

  - `raii.cpp`
  - `sharedPtrDeleter.cpp`
  - `lifetimeSemantic.cpp`

- Übungen:
  - Schreiben Sie eine einfache Lock wie `std::lock_guard`, der sich um seinen Mutex kümmert.
    - Lösung: `myGuard.cpp`
  - Erstellen Sie 100 Millionen `std::shared_ptr` mit `std::shared_ptr` und `std::make_shared`. Messen Sie die Performanz
    - Lösung: `sharedPtrPerformanz.cpp` und `makeSharedPerformanz.cpp`

Ressource Management

- Analysieren Sie das Programm `lifetimeSemantic.cpp`.
  - Übersetzen und führen Sie das Programm aus.
  - Warum ist die Funktion `asSmartPointerBad` schlecht?

- Angenommen, Sie haben die folgende Funktion:

```
void shared(std::shared_ptr<Widget>& shaPtr){
    oldLongRunningFunc(*shaPtr);
}
```

  - Wie können Sie sicherstellen, dass die dem `shaPtr` zugrundeliegende Ressource während des Aufrufs der Funktion `oldLongRunningFunc` gültig bleibt?

- Weitere Informationen:
  - Ressource management

Expressions und Statements

Expressions und Statements sind die niedrigste und direkteste Art, Aktionen und Berechnungen auszudrücken.

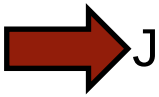
- Eine **Expression** wird zu einem Wert ausgewertet.
- Ein **Statement** tut etwas und setzt sich oft aus Expressions oder Statements zusammen.

```
5 * 5;              // expression
std::cout << 25; // print statement
auto a = 10;        // assignment statement
auto b = 5 * 5;  // expression statement
```

Gute Namen

Eine **Deklaration** ist eine Erklärung, die einen Namen in einen Bereich einführt.

- Gute Namen sind wohl die wichtigste Regel für gute Software.

- Gute Namen sollten
  - selbsterklärend sein.  Je kürzer der Anwendungsbereich, desto kürzer der Name.
  - nicht in verschachtelten Geltungsbereichen wiederverwendet werden.
  - sollten ähnlich aussehende Namen vermeiden.

```
if (readable(i1 + l1 + ol + o1 + o0 + I0 + l0)) {
    surprise();
}
```

Gute Namen

- so lokal wie möglich sein.

```
std::map<int,std::string> myMap;
if (auto result = myMap.insert(value); result.second) {
  useResult(result.first);
}
else {
} // result is automatically destroyed
```

- nicht aus `All_CAPS` bestehen.

```
#define NE !=                          // somewhere in a header
enum Coord { N, NE, NW, S, SE, SW}; // in another header
switch (direction) {                   // in some cpp
case N:
    // ...
case NE:
   // ...
}
```

Gute Namen

- ausschließlich exklusive pro Zeile deklariert werden.

```
char* p, p2;
char a = 'a';
p = &a;
p2 = a;
int a = 7, b = 9, c, d = 10, e = 3;
```

`auto:` Erhalte nie den falschen Datentyp

```
auto a= 5;
auto b= 10;
auto sum =  a * b * 3;
auto res = sum + 10;
std::cout << typeid(res).name();          // i

auto a2 = 5;
auto b2 = 10.5;
auto sum2 = a2 * b2 * 3;
auto res2 = sum2 * 10;
std::cout << typeid(res2).name();         // d

auto a3 = 5;
auto b3 = 10;
auto sum3 = a3 * b3 * 3.1f;
auto res3 = sum3 * 10;
std::cout << typeid(res3).name();         // f
```

`auto`: Initialisiere immer

```
struct T1 {};

class T2{
 public:
    T2() {}
};

auto n = 0;

int main() {
    auto n2 = 0;
    auto s = ""s;
    auto t1 = T1();
    auto t2 = T2();
}
```

{}-Initialisierung

{}-Initialisierung ist

- immer anwendbar.
- löst den “*most vexing parse*” auf.
- verhindert die verengende Konvertierung (narrowing conversion).

- `auto`

Beispiel	C++11	C++17
`auto i{1};`	`std::initializer_list<int>`	`int`
`auto i = {1};`	`std::initializer_list<int>`	`std::initializer_list<int>`
`auto i{1, 2};`	`std::initializer_list<int>`	`ERROR`
`auto i = {1, 2};`	`std::initializer_list<int>`	`std::initializer_list<int>`

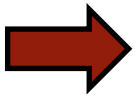
uniformInitialization.cpp
narrowingConversion.cpp

nullptr

Verwenden Sie `nullptr` anstatt von `0` oder `NULL`, um Zeiger zu initialisieren.

- **0**: Das Literal 0 kann der Null-Zeiger `(void*)0` oder die Zahl `0` sein.
- **NULL**: NULL ist ein Makro. Daher wissen Sie nicht, was drinsteht.
- Eine mögliche Implementierung (cppreference.com):

```
#define NULL 0
//since C++11
#define NULL nullptr
```

 Der Null-Zeiger `0` oder das Makro `NULL` sind ungeeignet für generischen Code.

Casts

- Falls notwendig, verwenden Sie benamste Casts:
  - **`static_cast`**: konvertiert ähnliche Typen wie Zeiger oder Zahlen
  - **`const_cast`**: fügt `const` oder `volatile` hinzu oder entfernt es
  - **`reinterpret_cast`**: konvertiert Zeiger oder Integrale und Zeiger
  - **`dynamic_cast`**: konvertiert polymorphe Zeiger oder Referenzen in die gleiche Klassenhierarchie
  - **`std::move`**: konvertiert in eine Rvalue-Referenz
  - **`std::forward`**: konvertiert einen Lvalue zu einer Lvalue-Referenz und ein Rvalue zu reiner Rvalue-Referenz

- Entfernen Sie `const` nicht von einem ursprünglichen `const` Object

```
const int constInt = 10;
const int* pToConstInt = &constInt;
int* pToInt = const_cast<int*>(pToConstInt);
*pToInt = 12;     // undefined behavior
```

Statements

- Bevorzugen sie Algorithmen der STL gegenüber Schleifen.

```
std::vector<int> vec = {-10, 5, 0, 3, -20, 31};
std::sort(std::execute::par, vec.begin(), vec.end());
std::sort(std::execute::par_unseq, vec.begin(), vec.end())
```

- Bevorzugen Sie range-basierte for-loops gegenüber for-loops; bevorzugen Sie for-loops gegenüber while-loops.
- Verlassen Sie sich nicht auf impliziertes fallthrough in switch statements.
- Benutzen Sie `[[fallthrough]]` um anzuzeigen, dass der fallthrough beabsichtigt ist.

```
switch (n) {
    case 1:
        g();
        [[fallthrough]];
    case 2:
        h();
}
```

Arithmetic

- Mischen Sie keine vorzeichenlose und vorzeichenbehaftete Arithmetik.

```
#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
}
```

Expressions und Statements

- Beispiele:
  - `shadowClass.cpp`
  - `uniformInitialization.cpp`
  - `mostVexingParse.cpp`
  - `narrowingConversion.cpp`
  - `nullPointer.cpp`
  - `unspecified.cpp`
  - `strange1.cpp` und `strange2.cpp`
  - `overUnderflow.cpp`

- Übungen:
  - Warum schlägt das Übersetzen des Programms `shadowClass.cpp` fehl? Beheben Sie den Fehler
    - Lösung: `shadowClass.cpp`

Expressions und Statements

- Übungen:

  - Beheben Sie den Fehler in dem Programm `mostVexingParse.cpp`.
    - Lösung: `mostVexingParserSolved.cpp`
  - Beheben Sie den Fehler in dem Programm `narrowingConversion.cpp`.
    - Lösung: `narrowingConversionSolved.cpp`
  - Führen Sie das Programm `unspecified.cpp` auf GCC und clang aus. Welcher Compiler ist korrekt?

Expressions und Statements

- Übungen:
  - Das Programm `strange1.cpp` und `strange2.cpp` produziert unterschiedliche Ergebnisse. In dem Programm `strange1.cpp` ist die Summenvariable `x` ein `unsigned short` und in `strange2.cpp` `x` ist sie ein `short`. Was ist passiert?
  - Wie lang läuft das Programm `overUnderflow.cpp` auf ihrer Plattform? Probieren Sie es aus.
- Weitere Informationen:
  - Expressions und statements

Performanz

Falsche Optimierung

- “premature optimization is the root of all evil” (Donald Knuth)

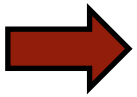
- Regeln der Optimierung
  1. Messen Sie ihr Programm mit repräsentativen Daten.
  2. Entwickeln Sie eine Intuition, welche Performanz maximal möglich ist.
  3. Versionieren Sie Ihren Performanz-Test.

- Bedeutung von Messungen
  - Welcher Teil des Programms ist der Engpass?
  - Wie schnell ist gut genug für den Benutzer?
  - Wie schnell könnte das Programm potentiell sein?

Performanz

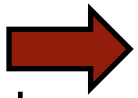
Falsche Annahmen

- Gehen Sie nicht davon aus, dass komplizierter Code notwendigerweise schneller ist als einfacher Code.
- Gehen Sie nicht davon aus, dass Low-Level-Code notwendigerweise schneller ist als High-Level-Code.
- Machen Sie keine Behauptungen über Performanz ohne Messungen.

 Finden Sie mit dem Compiler-Explorer die ultimative Wahrheit über den optimierten Code heraus.

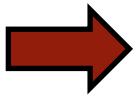
Performanz

Behalten Sie beim Programmieren die Optimierung im Fokus.

- Benutzen Sie - wenn möglich - Move-Semantik
  - Benutzen Sie billige Move-Semantik anstatt aufwendiger Copy-Semantik.
  - keine Speicheranforderung notwendig  keine `std::bad_alloc` Ausnahmen möglich
  - Sie können Datentypen verwenden, die nur Move-Semantik unterstützen: `std::unique_ptr`.
- Verlassen Sie sich auf das statische Typensystem.
  - lokalen Code schreiben
  - einfachen Code schreiben
  - geben Sie dem Compiler zusätzliche Hinweise(`noexcept`, `final`)

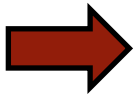
Performanz

- Wenn ihre Funktion zur Compilezeit ausgeführt werden kann, implementiere sie als `constexpr`.



Eine `constexpr` Funktion kann zur Compilezeit und zur Laufzeit ausgeführt werden.

- Respektieren Sie Cache-Lines
  - Wenn sie ein `int` aus dem Speicher lesen, wird typischerweise eine Cache-line von 64 Bytes (16 `int`'s) ausgelesen und einen schnellen Cache gespeichert.



Zusammenhänge Speicherbereiche lesen respektiert Cache-Lines.

Performanz

- Beispiele:

  - `singletonAcquireRelease.cpp`
  - `singletonMeyers.cpp`
  - `memoryAccess.cpp`

- Übungen:
  - Messen Sie die Performanz der Programme `singletonMeyers.cpp` und `singletonAcquireRelease.cpp`.
    - Welches Programm ist schneller?
    - Welche Performanz können Sie maximal erhalten?

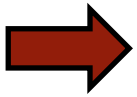
Performanz

- Messen die Performanz des Programmes `memoryAccess.cpp`. Haben Sie diese Zahlen erwartet?
- Die geordneten assoziativen Container wie `std::map` haben kein Layout, das Cache-Lines respektiert. Implementieren Sie eine `std::map`, die Cache-Lines respektiert.
  - Lösung: `flatMap.cpp`

- Weitere Informationen:
  - Performanz

Concurrency und Parallelität

Locks

- NNM (**N**o **N**aked **M**utex)  Verwalten Sie das Mutex in einem Objekt: Lock.
- Verwenden Sie `std::lock` or `std::scoped_lock` um atomare Mutexe zu locken.

- Geben Sie Locks einen Namen.

```
{
    std::lock_guard<std::mutex> {m};
    std::cout << "CRITICAL SECTION" << std::endl;
}
```

Concurreny und Parallelität

Threads

- Bevorzugen Sie `std::jthread` gegenüber `std::thread`.
- *Detachen* Sie keinen `thread`.
- Übergeben Sie kleine Datenmengen zwischen Threads per Copy.
- Verwenden Sie `std::shared_ptr` um Besitzverhältnisse zwischen unabhängigen Threads zu koordinieren.

`threadDetach.cpp`

Concurreny und Parallelität

Condition-Variablen

- Warten Sie nicht ohne eine Bedingung; achten Sie auf *spurious wakeups* und *lost wakeups*.
- Wenn möglich, bevorzugen Sie Tasks (Promise und Futures) um Thread zu synchronisieren.

Kriterium	Bedingungsvariable	Tasks
Mehrfache Synchronisierung	Ja	Nein
Kritischer Bereich	Ja	Nein
Spurious wakeup	Ja	Nein
Lost wakeup	Ja	Nein

`conditionVariable.cpp`
`promiseFutureSynchronize.cpp`

Concurrency und Parallelität

- Benutzen Sie jedes verfügbare Werkzeug, um Ihren concurrent Code zu validieren.
  - ThreadSanitizer
    - dynamischer Code-Analysierer
    - Teil von clang 3.2 und GCC 4.8
    - übersetzen Sie das Programm mit –fsanitize=thread -g -O2.
  - CppMem
    - statischer Code-Analysator
    - validiert kleine Code-Schnipsel, die typischerweise atomare Variablen beinhalten
    - gibt Ihnen einen tiefen Einblick in das C++-Speichermodell

Concurrency und Parallelität

Parallelität

- Bevorzugen Sie die parallelen Algorithmen der STL gegenüber selbstimplementieren Lösungen, die auf Threads basieren.

```
std::vector<int> v = {5, -3, 10, -5, -10, 22, 0};

std::sort(v.begin(), v.end());                                  // sequential

std::sort(std::execution::seq, v.begin(), v.end());             // sequential
std::sort(std::execution::par, v.begin(), v.end());             // parallel
std::sort(std::execution::par_unseq, v.begin(), v.end()); // vectorized
```

transformExclusiveScan.cpp

Concurrency und Parallelität

Message Passing

- Denken Sie in Aufgaben (Promise/Future Paaren) und nicht in Threads.
- Verwenden Sie ein Future, um einen Wert oder eine Ausnahme von einer gleichzeitigen Aufgabe zu erhalten.

`promiseFutureException.cpp`

Concurrency und Parallelität

Atomics

- Programmieren Sie nur für sehr einfache Aufgaben lock-frei.
- Trauen Sie nicht Ihrer Intuition.
- Studieren Sie sorgfältig die Literatur, bevor Sie das Programm lock-frei programmieren.

- **Herb Sutter**: "*Lock-free programming is like playing with knives ".*
- **Anthony Williams**: "*Lock-free programming is about how to shoot yourself in the foot.*"
- **Tony Van Eerd**: "*Lock-free coding is the last thing you want to do*."
- **Fedor Pikus**: "*Writing lock-free programs is hard. Writing correct lock-free programs is even harder.*"
- **Harald Böhm**: "*The rules are not obvious.* "

`sequentialConsistency.cpp`
`relaxedSemantic.cpp`

Concurrency und Parallelität

- Beispiele:

  - `threadDetach.cpp`
  - `threadSharesOwnership.cpp`
  - `conditionVariable.cpp`
  - `transformExclusiveScan.cpp`
  - `promiseFutureException.cpp`
  - `promiseFutureSynchronize.cpp`
  - `sequentialConsistency.cpp`
  - `relaxedSemantic.cpp`

Concurrency und Parallelität

- Übungen:

  - Warum ist der folgende Code-Schnipsel sehr schlecht?

```
std::mutex m;
m.lock();
sharedVariable = unknownFunction();
m.unlock();
```

  - Was ist das Problem des Programmes `threadDetach.cpp`? Beheben Sie das Problem.

  - Die Threads des Programmes `threadSharesOwnership.cpp` teilen die Variable `tmpInt`. Benutzen Sie einen `std::shared_ptr` um die Ressource zwischen den Threads zu teilen.
    - Lösung: `threadSharesOwnershipSmartPtr.cpp`

Concurrency und Parallelität

- Schreiben Sie ein einfaches Tischtennisspiel.
  - Zwei Threads sollten abwechselnd ein `bool` value auf `true` oder `false` setzen. Ein Thread setzt den Wert auf `true` und benachrichtigt den anderen Thread. Der andere Thread setzt den Wert auf `false` und benachrichtigt den ursprünglichen Thread. Dieses Spiel sollte nach einer festgelegten Anzahl von Iterationen enden.
    - Lösung: `conditonVariablePingPong.cpp`

- Weitere Informationen:
  - Concurrency
  - Anthony Williams: C++ Concurrency in Action. Manning Publications
  - Bartosz Milewski: Bartosz Milewski's Programming Cafe
  - Herb Sutter: Effective Concurrency
  - Jeff Preshing: Preshing on Programming

Error Handling

Error Handling besteht aus

- erkennen des Fehlers.
- übertragen von Informationen zu einem Fehler an einen Handler.
- den definierten Zustand des Programms beibehalten.
- vermeiden von Ressourcelecks.

Abrahams Garantien

- No-throw guarantee
- Strong exception safety (Rollback Semantik)
- Basic exception safety (Invarianten erhalten)
- No exception safety

Error Handling

- Jede Softwareeinheit hat zwei Kanäle:
  - Regulärer Kanal: Was die Software-Einheit tun soll.
  - Irregulärer Kanal: Wie die Software-Einheit arbeiten soll.
- Bauen Sie Ihr Error Handling auf Invarianten auf. Wenn diese Invarianten nicht mehr erfüllt werden können, werfen Sie eine Ausnahme.
- Verwenden Sie benutzer-definierte Typen für Ausnahmen.
- Fangen Sie Ausnahmen vom Speziellen zum Allgemeinen.
- Verwenden Sie Ausnahmen nur für die Ausnahmebehandlung.
- Werfen Sie niemals, solange Sie ein direkter Besitzer sind.

Error Handling

- Übungen:

  - Kennen Sie falsche Einsätze von Ausnahmen?
  - Welches Problem besitzt der folgende Code?

```
int getIndex(std::vector<std::string>& vec,
             const std::string& x) {
    try {
        for (auto i = 0; i < vec.size(); ++i)
            if (vec[i] == x) throw i; // found x
    } catch (int i) {
        return i;
    }
    return -1; // not found
}
```

Error Handling

- Welches Problem besitzt der folgende Code?

```
void leak(int x) {
    auto p = new int{7};
    if (x < 0) throw Get_me_out_of_here{};
    // ...
    delete p; // we may never get here
}
```

- Weitere Informationen:
  - Error handling
  - David Abrahams: Exception-Safety in Generic Components

Konstanten und Immutabilität

- Per Default sollten Objekte konstant sein.
  - kann nicht Opfer eines Data-Races werden
  - sichern Sie zu, dass sie thread-sicher initialisiert werden
  - Unterscheiden Sie zwischen physischer und logischer Konstantheit.

- Das Entfernen von `const` eines als `const` erzeugten Objekts ist undefiniertes Verhalten, wenn dieses verändert wird.

`mutable.cpp`
`constCastAway.cpp`

Konstanten und Immutabilität

- Per default sollten Funktionen konstant sein

- Wenn möglich, machen Sie ihre Funktionen `constexpr`.
  - bietet zusätzliche Optimierungsmöglichkeiten
  - `constexpr` Funktionen sind nahezu rein

Konstanten und Immutabilität

- Beispiele:
  - `mutable.cpp`
  - `constCastAway.cpp`
- Übungen:
  - Implementieren Sie eine Klasse `Lock` mit zwei konstanten Methoden `lock` und `unlock`. Die Klasse `Lock` sollte einen `std::mutex` haben, welcher das Locken durchführt.
    - Lösungs: `strategizedLockingCompileTime.cpp`
  - Eine `constexpr` Funktion kann zur Laufzeit laufen. Wie können Sie prüfen, ob eine `constexpr` Funktion zur Compilezeit oder Laufzeit ausgeführt wird?
- Weitere Informationen:
  - Konstanten und Immutabilität

Templates

Einsatz

- Verwenden sie Templates zur Erhöhung des Abstraktionsniveaus.
- Verwenden Sie Templates, um Algorithmen auszudrücken, die für viele Datentypen gelten.

Interfaces

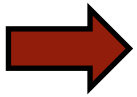
- Verwenden Sie Funktionsobjekte (Lambdas), um Operationen an Algorithmen zu übergeben.
- Lassen Sie den Compiler die Template-Argumente bestimmen.
- Template-Argumente sollten mindestens `SemiRegular` oder `Regular` sein.

`notGeneric.cpp`
`functionObjects.cpp`
`templateArgumentDeduction.cpp`

Templates

- **Argument-dependent lookup:** Unqualifizierter Funktionsnamen werden zusätzlich im Namensraum ihrer Argumente nachgeschlagen.

```
#include <iostream>
int main() {
    std::cout << "Argument-dependent lookup";
}
```



Warum wird der `operator<<` gefunden?

- Concepts können mit `std::enable_if` simuliert werden.

`enable_if.cpp`

Templates

Definition

- Platzieren Sie non-dependent Mitglieder eines Klassen-Templates in einer Basisklasse, die kein Template ist.
- Es gibt verschiedene Möglichkeiten, so dass ein benutzer-definierter Datentyp `MyType` eine generischen Funktion `isSmaller` unterstützt.

```
template <typename T>
bool isSmaller(T fir, T sec) {
    return fir < sec;
}
```

  - Implementieren Sie `operator <` für `MyType`.
  - Implementieren Sie eine volle Spezialisierung für `MyType`.
  - Erweitern Sie `isSmaller` um ein Prädikat.

`isSmaller.cpp`

Templates

Hierarchien

- Virtuelle Funktionen werden jedes Mal in einem Klassen-Template instanziiert. Im Gegensatz dazu werden nicht-virtuelle Funktionen nur bei Bedarf instanziiert.
- Generische Methoden können nicht virtuell sein.

Variadic Templates

- Fabrikfunktionen in modernem C++ beruhen auf zwei mächtigen Features: Perfect Forwarding und Variadic Templates.
- Dank Perfect Forwarding und Variadic Templates kann eine Factory-Funktion eine beliebige Anzahl von L- und R-Values annehmen.

`virtualNonVirtualMemberFunctionTemplates.cpp`
`perfectForwarding.cpp`

Templates

Metaprogrammierung

- Metaprogrammierung ist Programmierung zur Compilezeit.
- Metaprogrammierung kann mit Templates, der Type-Traits Bibliothek oder `constexpr` Funktionen umgesetzt werden.
- Bevorzugen Sie `constexpr` Funktionen gegenüber der Type-Traits Bibliothek; bevorzugen Sie die Type-Traits Bibliothek gegenüber der Template Metaprogrammierung.

Weitere Regeln

- Verwenden Sie ein Lambda, wenn Sie eine einfache Operation an Ort und Stelle benötigen.
- Geben Sie Operationen mit dem Potenzial zur Wiederverwendung einen Namen.
- Schreiben Sie nicht unbeabsichtigt nicht generischen Code.

Templates

- Beispiele:
  - `templateArgumentDeduction.cpp`
  - `semiRegular.cpp`
  - `argumentDependentLookup.cpp`
  - `enable_if.cpp`
  - `isSmaller.cpp`
  - `genericArray.cpp`
  - `genericArrayVererbung.cpp`
  - `virtualNonVirtualMemberFunctionTemplates.cpp`
  - `perfectfürwarding.cpp`
  - `records.cpp`
  - `notGeneric.cpp`
  - `functionObjects.cpp`

Templates

- Übungen:

  - Wie können die folgenden Code-Schnipsel verbessert werden?

```
template<typename T>
    requires Incrementable<T>
T sum1(vector<T>& v, T s) {
    für (auto x : v) s += x;
    return s;
}

template<typename T>
    requires Addable<T>
T sum2(vector<T>& v, T s) {
    für (auto x : v) s = s + x;
     return s;
 }
```

Templates

- Der Gleichheits-Operator in `argumentDependentLookup.cpp` ist zu sichtbar. Beheben Sie das Problem.
  - Lösung: `argumentDependentLookupResolved.cpp`

- Studieren Sie den Instanziierungsprozess von `Array` in den Programmen `genericArray.cpp` und `genericArrayInheritance.cpp`. Benutzen Sie C++ Insights für ihre Studie.

- Erweitern Sie den benutzer-definierten Typ `Account` (`isSmaller.cpp`) so dass Instanzen von `Account` anhand des Saldos verglichen werden können. Diskutieren Sie die Vor- und Nachteile der verschiedenen Lösungen. Lösung: `accountIsSmaller1.cpp`, `accountIsSmaller2.cpp`, `accountIsSmaller3.cpp`

Templates

- Studieren Sie das Programm `perfectForwarding.cpp` in C++ Insights.
- Refaktorieren Sie das Programm `records.cpp`. Geben Sie dem Prädikat für den case-insensitiven Vergleich einen Namen und verwenden Sie diesen.
  - Lösung: `records.cpp`

- Weitere Informationen:
  - Templats und Generische Programmierung

C-Style Programmierung

- Wenn Sie C verwenden müssen, übersetzen Sie den C-Code mit C++.
  - Vollständiger Sourcecode verfügbar: Fertig.
  - Gesamter Soucecode nicht verfügbar:
  1. Verwenden Sie Ihren C++-Compiler, um Ihre Hauptfunktion zu übersetzen.
  2. Benutzen Sie Ihren C++-Compiler, um Ihr Programm zu linken.
  3. Verwenden Sie einen C- und C++-Compiler des gleichen Herstellers, der die gleichen Aufrufkonventionen haben sollte.
- Wenn Sie C für Schnittstellen verwenden müssen, unterdrücken Sie das Namen Mangeling.
  - Der C++-Compiler kodiert zusätzlich den Typ der Parameter in die Funktionsnamen.

    ➡ Überladen von Funktionen

Funktion	GCC 8.3	MSVC 19.16
`print(i)`	`_Z5Printi`	`?print@@YAXH@Z`
`print(double)`	`_Z5Printd`	`?print@@YAXN@Z`

C-Style Programmierung

Das Name Mangling unterdrücken.

- Jede Funktion

```
extern "C" void
foo(int);
```

- Jede Funktion in einem Scope

```
extern "C" {
    void foo(int);
    double bar(double);
};
```

- Jede Funktion in einem Header

```
#ifdef __cplusplus
extern "C" {
#endif
    void foo(int);
    double bar(double);
    ...
#ifdef __cplusplus
}
#endif
```

C-Style Programmierung

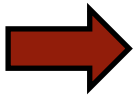
- Beispiele:
  - `functionOverloading.cpp`

- Übungen:
  - Benutzen Sie den Compiler-Explorer, um die verschiedenen Möglichkeiten der verschiedenen Compiler im Bereich des Name Mangeling zu visualisieren. Beantworten Sie die folgenden Fragen mit `functionOverloading.cpp`:
    - Kann sich das Name-Mangeling zwischen den Compiler-Versionen verändern?
    - Hat die Cross-Compilierung Auswirkungen auf das Name-Mangeling?
- Weitere Informationen:
  - C-style Programmierung

Quelldateien

- Eine Header-Datei sollte weder eine Objektdefinition noch die Definition einer nicht-`inline` Funktion enthalten.
  - ODR (One Definition Rule)
    - Eine Funktion kann in einer Übersetzungseinheit nicht mehr als einmal definiert werden.
    - Eine Funktion kann kann in einem Programm nicht mehr als einmal definiert werden.
    - Inline-Funktionen können in mehr als einer Übersetzungeinheit definiert werden. Die Definitionen müssen gleich sein.

- Vermeiden Sie zyklische Abhängigkeiten zwischen Sourcecodedateien



Namen sollten vorwärts deklariert werden.

Quelldateien

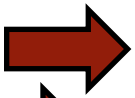
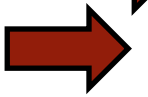
- Beispiele:
  - `impl.cpp, impl.h, odr.cpp`
  - `dependency.cpp, a.h, b.h`

- Übungen:
  - Kompilieren Sie das Programm `odr`, das aus den Dateien `impl.cpp`, `impl.h` und `odr.cpp` in zwei Varianten.
    - Verwenden Sie in der Datei `impl.cpp`. nicht die Headerdatei `impl.h`
    - Verwenden Sie in der Datei `impl.cpp`. die Headerdatei `impl.h`
    - Sind die Fehlermeldungen auf den Compiler oder auf den Linker zurückzuführen?

Quelldateien

- Übungen:
  - Das Programm `dependency.cpp` hat eine zirkuläre Abhängigkeit. Wie können Sie dieses Problem lösen?
    - Lösung: `dependency.cpp, a.h und b.h`
- Weitere Informationen:
  - Quelldateien

Die Standardbibliothek

- Ziehen Sie `std::array` und `std::vector` gegenüber einem C-array vor.
  - Die Containergröße ist bekannt  `std::array`
  - Der Container sollte groß sein  `std::vector`

  - `std::vector` und `std::array`

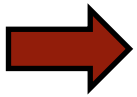
    - kennen ihre Größe.
    - verwalten automatisch ihren Speicher (RAII).
    - erlauben den geschützten Elementzugriff mittels des `at`-Operators.
    - besitzen ein ideales Speicherlayout.

`vectorMemory.cpp`

Die Standardbibliothek

Die acht Varianten von assoziativen Containern in C++.

Associative Container	Schlüssel sortiert	Wert	Mehrere gleiche Schlüssel	Zugriffszeit
`std::set`	Ja	Nein	Nein	Logarithmisch
`std::unordered_set`	Nein	Nein	Nein	Konstant (Durchschnitt)
`std::map`	Ja	Ja	Nein	Logarithmisch
`std::unordered_map`	Nein	Ja	Nein	Konstant (Durchschnitt)
`std::multiset`	Ja	Nein	Ja	Logarithmisch
`std::unordered_multiset`	Nein	Nein	Ja	Konstant (Durchschnitt)
`std::multimap`	Ja	Ja	Ja	Logarithmisch
`std::unordered_multimap`	Nein	Ja	Ja	Konstant (Durchschnitt)



Bevorzugen Sie `std::unordered_map` anstelle von `std::map` aus Performanz-Gründen.

Die Standardbibliothek

- Text
  - Benutzen Sie `std::string` für eigene Charakter-Sequenzen.
  - Benutzen Sie `std::string_view` um eine Charakter-Sequenz zu referenzieren.
    - Es unterstützt hauptsächlich Leseoperationen, die dem Interface einer `std::string` folgen.
    - Es besitzt die zwei modifizierenden Operationen `remove_prefix` und `remove_suffix`.
  - Nutzen Sie `char*` um sich auf einen einzelnen Charakter zu beziehen.
  - Nutzen Sie `std::byte` um auf Bytes zu verweisen, die keinen Charakter darstellen.
    - Es unterstützt nur die bitweisen logischen Operationen.

      `<<`, `>>`, `|`, `&`, `~` und `^`

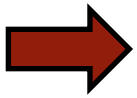
`stringView.cpp`

Die Standardbibliothek

- In- und Output
  - Berücksichtigen Sie beim Lesen immer fehlerhafte Eingaben.

Flag	Abfrage	Beschreibung
`std::ios::goodbit`	`stream.good()`	Kein Bit gesetzt
`std::ios::eofbit`	`stream.eof()`	end-of-file Bit gesetzt
`std::ios::failbit`	`stream.fail()`	Fehler
`std::ios::badbit`	`stream.bad()`	Undefined Verhalten

  - Prefer iostreams für I/O



Der Compiler bestimmt automatisch den Datentyp.

`streamState.cpp`
`printfIostreams.cpp`

Die Standardbibliothek

- Reguläre Ausdrücke
  - Nutzen Sie nur reguläre Ausdrücke, wenn es notwendig ist.
    - Prüfen, ob ein Text mit einem Textmuster übereinstimmt: `std::regex_match`.
    - Ein Textmuster im Text suchen: `std::regex_search`.
    - Ein Textmuster mit einem Text ersetzen: `std::regex_replace`.
    - Über alle Textmuster in einem Text iterieren: `std::regex_iterator` und `std::regex_token_iterator`.
    - Das `match_result` für eine weitere Analyse verwenden.
  - Raw-Strings sollten für reguläre Ausdrücke verwendet werden.

```
std::string regExpr("C\\+\\+");
std::string regExprRaw(R"(C\+\+)");
```

regexSearchEmail.cpp
regexSearchFloatingPoint.cpp

Die Standardbibliothek

- Reguläre Ausdrücke
  - Der Text bestimmt den Typ des regulären Ausdrucks, des Ergebnisses und der Erfassungsgruppe.

Text	Regulärer Ausdruck	Ergebnis	Erfassungsgruppe
`const char*`	`std::regex`	`std::cmatch`	`std::csub_match`
`std::string`	`std::regex`	`std::smatch`	`std::ssub_match`
`std::wchar_t*`	`std::wregex`	`std::wcmatch`	`std::wcsub_match`
`std::wstring`	`std::wregex`	`std::wsmacht`	`std::wssub_match`

  - Nutzen Sie `std::regex_iterator` oder `std::regex_token_iterator` für wiederholtes Suchen.

search.cpp

Die Standardbibliothek

- Beispiele:

  - `vectorMemory.cpp`
  - `memoryAccess.cpp`
  - `overUnderflowStdArray.cpp`
  - `overUnderflowStdVector.cpp`
  - `stringView.cpp`
  - `streamState.cpp`
  - `printfIostreams.cpp`
  - `printfIostreamsUndefinedBehaviour.cpp`
  - `regexSearchFloatingPoint.cpp`
  - `regexSearchEmail.cpp`
  - `search.cpp`
  - `wordCount.cpp`

Die Standardbibliothek

- Übungen:
  - Vergleich Sie die Sequenz-Container. Kennen Sie einen Grund `std::array` oder `std::vector` nicht zu verwenden?

Charakteristiken	`std::array`	`std:: vector`	`std::deque`	`std::list`	`std::forward_list`
Größe	Static	Dynamisch	Dynamisch	Dynamisch	Dynamisch
Implementierung	Static Array	Dynamisches Array	Verkettete Arrays	Doppelt verkettete Liste	Einfach verkettete Liste
Zugriff	Wahlfreier Zugriff	Wahlfreier Zugriff	Wahlfreier Zugriff	Vorwärts und rückwärts	Forwärts
Optimiert für		Ende O(1)	Beginn und Ende O(1)	• Beginn und Ende O(1) • Beliebig O(1)	• Beginn O(1) • Beliebig O(1)
Explizite Speicheranforderung		Ja	Nein	Nein	Nein
Explizite Speicherfreigabe		`shrink_to_fit()`	• manchmal • `shrink_to_fit()`	Immer	immer
Stärken	• keine Speicher-anforderung • kein Overhead	95% Lösung	Hinzufügen und Löschen am Beginn und Ende	Hinzufügen und Löschen an einer beliebigen Stelle	• schnelles Hinzufügen und Löschen • minimale Speicher-anforderungen
Schwächen	Keine dynamische Speicheranforderung	Hinzufügen oder Löschen an beliebiger Stelle 0(n)	Hinzufügen oder Löschen an beliebiger Stelle O(n)	Kein wahlfreier Zugriff	Kein wahlfreier Zugriff

search.cpp

Die Standardbibliothek

- Übungen:

  - Führen sie das Programm `memoryAccess.cpp` aus. Vergleichen Sie die Performanzzahlen der verschiedenen Container.
  - Führen Sie die Programme `overUnderflowStdArray.cpp` und `overUnderflowStdVector.cpp` aus. Wie lang geht dies gut?
  - Führen Sie das Programm `printfIostreamsUndefinedBehaviour.cpp` aus. Verwenden Sie die auskommentierten Zeilen.
  - Studieren Sie das Programm `wordCount.cpp`. Analysieren Sie die Datei `grimm.txt` mit Hilfe des Programms.

Die Standardbibliothek

- Weitere Informationen:

  - Die Standardbibliothek
  - More special Friend with `std::map` and `std::unordered_map`
  - C++17 – Avoid copying with `std::string_view`
  - Stuff you should know about In- und Output with Streams

Clean Code in C++

- Wichtige Grundsätze
- Best Practices
- **Pattern und Idiome**
- Testen
- Refactoring
- Werkzeugunterstützung

Definition

"*Each pattern is a three-part rule, which express a relation between a certain context, a problem und a solution*."
(Christopher Alexander)

- Die Klassiker:
  - Design Patterns: Elements of Reusable Object-Oriented Software von Eric Gamma, Richard Helm, Ralph Johnson und John Vlissides ("Gang of Four", GOF); 1994
  - Pattern-Oriented Software Architecture: A System of Patterns von Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal ("**P**attern**-O**riented **S**oftware **A**rchitecture", POSA); 1996

Vererbung versus Komposition

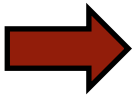
Die Klasse `Car` steht für ein generisches Auto.

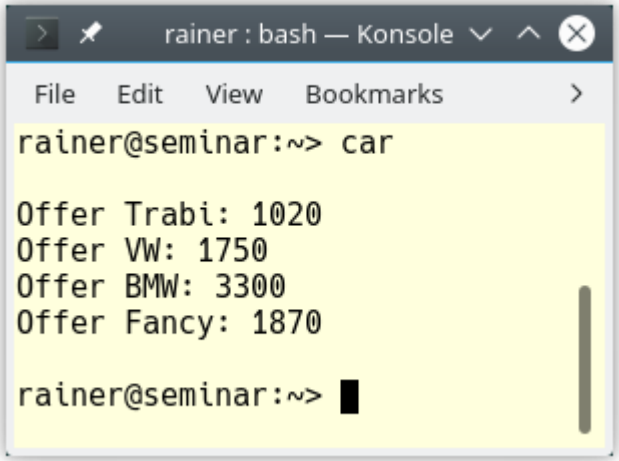
- `Car` besteht aus vier Rädern, einem Motor und einer Karosserie.
  - der Preis des Autos ist die Summe aller seiner Teile
  - gibt den Preis zurück, indem es die Abfrage an alle seine Teile delegiert

```
struct Car{
    Car(std::unique_ptr<Wheel> wh, std::unique_ptr<Motor> mo, std::unique_ptr<Body> bo):
        myWheel(std::move(wh)), myMotor(std::move(mo)), myBody(std::move(bo)){}

    int getPrice(){
        return 4 * myWheel->getPrice() + myMotor->getPrice() + myBody->getPrice();
    }

private:
    std::unique_ptr<Wheel> myWheel;
    std::unique_ptr<Motor> myMotor;
    std::unique_ptr<Body> myBody;

};
```

Vererbung versus Komposition

```
int main(){

    std::cout << std::endl;

    Car trabi(std::make_unique<TrabiWheel>(), std::make_unique<TrabiMotor>(), std::make_unique<TrabiBody>());
    std::cout << "Offer Trabi: " << trabi.getPrice() << std::endl;

    Car vw(std::make_unique<VWWheel>(), std::make_unique<VWMotor>(), std::make_unique<VWBody>());
    std::cout << "Offer VW: " << vw.getPrice() << std::endl;

    Car bmw(std::make_unique<BMWWheel>(), std::make_unique<BMWMotor>(), std::make_unique<BMWBody>());
    std::cout << "Offer BMW: " << bmw.getPrice() << std::endl;

    Car fancy(std::make_unique<TrabiWheel>(), std::make_unique<VWMotor>(), std::make_unique<BMWBody>());
    std::cout << "Offer Fancy: " << fancy.getPrice() << std::endl;

    std::cout << std::endl;

}
```





Vererbung versus Komposition

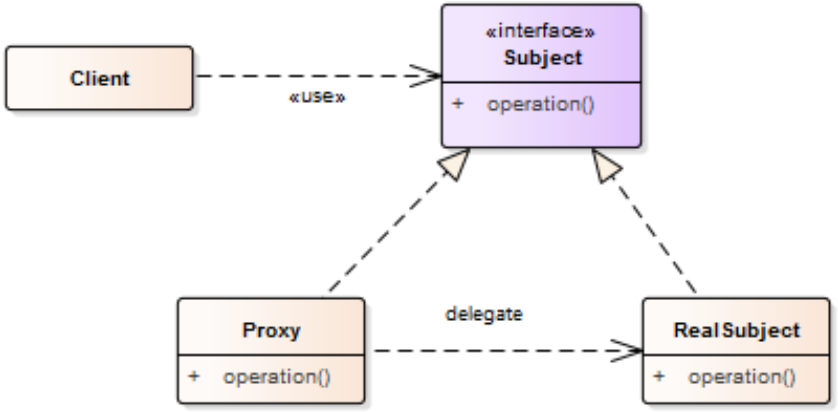
- Studieren Sie die Beispiele `car.cpp` und beantworten Sie die folgenden Fragen.

1. Wie viele verschiedene Autos können Sie aus den vorhandenen Fahrzeugkomponenten herstellen?
2. Wie viele Klassen benötigen Sie, um die gleiche Komplexität mit Vererbung zu lösen?
3. Wie einfach/schwierig ist es, mit Vererbung/Komposition ein neues Auto wie Audi zu unterstützen? Nehmen Sie dazu an, dass Ihnen alle Teile zu Ihrer Verfügung stehen.
4. Nehmen wir an, ein Kunde möchte ein neues, ausgefallenes Auto aus vorhandenen Fahrzeugkomponenten zusammenbauen lassen. Wann müssen Sie die Entscheidung treffen, das neue Auto auf Basis von Vererbung oder Komposition zu montieren? Welche Strategie wird zur Compilzeit und welche zur Laufzeit ausgeführt?

Proxy

- Typ
  - Strukturmuster
- Zweck
  - bietet einen Platzhalter für ein anderes Objekt, um den Zugriff darauf zu steuern
- Use-case
  - kontrolliert den Zugriff auf ein Objekt
    - Remote proxy (CORBA)
    - Virtual proxy (Projekt wird auf Anfrage erzeugt)
    - RAII

Proxy



- `Proxy`
  - kontrolliert den Zugriff und die Lebensdauer des `RealSubject`
  - unterstützt das gleiche Interface wie `RealSubject`
- `Subject`
  - definiert das gemeinsame Interface von `Proxy` und dem `RealSubject`
  - implementiert das Interface
- `RealSubject`
  - bietet die Implementierung an

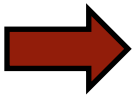
`proxy.cpp`

RAII

Resource **A**cquisition **I**s **I**nitialization (RAII).

RAII

- Erzeugt ein Proxy-Objekt für die Ressource.
- Der Konstruktor des Proxy fordert die Ressource an, der Destruktor des Proxy gibt die Ressource frei.
- Machen Sie den Proxy zu einem lokalen Objekt (Stack).



Die C++-Laufzeit kümmert sich um die Ressource.

raii.cpp

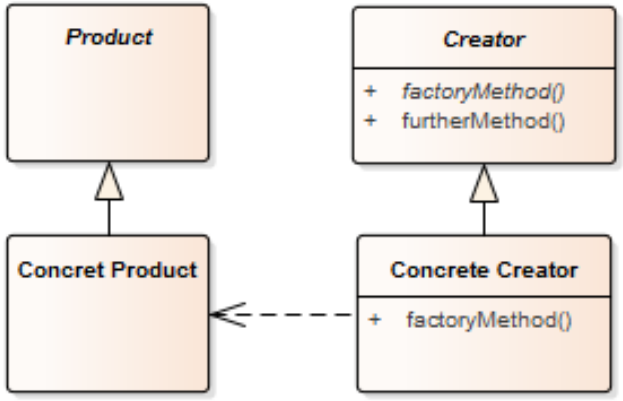
RAII

- Beispiele:
  - `raii.cpp`
  - `proxy.cpp`
- Übungen:
  - Das RAII-Idom wird in der C++-Standardbibliothek sehr stark genutzt. Nennen Sie bekannte Anwendungsfälle.
  - In welchen Situationen versagt RAII?
- Weitere Informationen:
  - Proxy
  - C++ Core Guidelines: When RAII breaks

Fabrikmethode

- Type
  - Erzeugermuster
- Zweck
  - Definieren Sie eine Schnittstelle zum Erstellen eines einzelnen Objekts, aber lassen Sie Unterklassen entscheiden, welche Klasse instanziiert werden soll.
- Auch bekannt als
  - Virtueller Konstruktor
- Anwendungsfall
  - Eine Klasse weiß nicht, welche Art von Objekten sie erzeugen soll.
  - Unterklassen entscheiden, welche Schnittstelle erstellt werden sollen.
  - Klassen delegieren die Erstellung von Objekten an Unterklassen weiter.

Fabrikmethode



- Product
  - von `factoryMethod` erstellte Objekte
- Concret Producte
  - implementiert die Schnittstelle
- Creator
  - deklariert die Fabrikmethode
  - ruft die Fabrikmethode auf
- Concrete Creator
  - überschreibt die Fabrikmethode

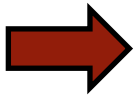
`factoryMethod.cpp`

Fabrikmethode

Eine klassische `Window` Fabrik.

```
class Window {
  public:
    virtual Window* clone() = 0;
    virtual ~Window() {};
};

class DefaultWindow:
      public Window {
private:
    DefaultWindow* clone();
};
```

```
Window* createWindow(Window& win){
  return win.clone();
}

int main(){
  DefaultWindow defWindow;
  const Window* defWindow1 =
              createWindow(defWindow);
}
```



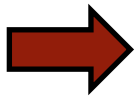
Probleme?

Fabrikmethode

```
int main(){
    DefaultWindow defWindow;
    const Window* defWindow1 = createWindow(defWindow);
}
```

Probleme:

- Wer ist der Besitzer von `Window`?
- Wer gibt die Ressource wieder frei?



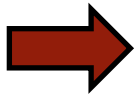
Benutzen Sie Smart Pointer

- `std::unique_ptr`: exklusiver Besitz
- `std::shared_ptr`: geteilter Besitz

Fabrikmethode

Die perfekte Fabrikmethode:

```
template <typename T, typename ... T1>
T create(T1&& ... t1) {
    return T(std::fürward<T1>(t1) ... );
}
```

 `create` steht für das Erzeuger-Idiom in modernem C++.

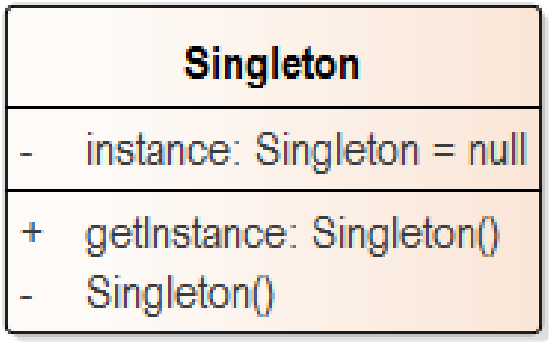
Fabrikmethode

- Beispiele:
  - `factoryMethod.cpp`
  - `factoryMethodUniquePtr.cpp`
- Übungen:
  1. Refaktorieren Sie die Implementierung der Fabrikmethode in `factoryMethodUniquePtr.cpp` zu `std::shared_ptr`.
     - Lösung: `factoryMethodSharedPtr.cpp`
  2. Machen Sie sich mit der Fabrikmethode `create` vertraut.
     - Wenden Sie die Funktion `create` an.
     - Analysieren Sie die Funktion mit C++ Insights.
     - Lösung: `perfectFactory.cpp`
- Weitere Informationen:
  - Fabrikmethode

Singleton

- Typ
  - Erzeugermuster
- Verwendung
  - Stellen Sie sicher, dass nur eine Instanz einer Klasse existiert.
- Anwendungsfall
  - Nur eine Instanz einer Klasse soll existieren.

Singleton



- `instance` (static)
  - private Instanz von `Singleton`
- `getInstance` (static)
  - öffentliche Methode, die `instance` zurückgibt
  - erzeugt `instance`
- `Singleton`
  - privater Konstruktor
  - das Aufrufen von `getInstance` ist die einzige Möglichkeit ein `Singleton` zu erzeugen.

`singleton.cpp`
`singletonMeyer.cpp`

Singleton

- Beispiele:
  - `singleton.cpp`
  - `singletonMeyer.cpp`

- Übungen:

  Das Singleton-Muster wird teilweise als Muster und teilweise als Anti-Muster gesehen. Diskutieren Sie die folgenden Fragen.

  1. Was sind die Vor- und Nachteile des Singleton-Musters?
  2. Kennen Sie Alternativen zum Singleton-Muster?

- Weitere Informationen:
  - Thread-Safe Initialisierung of a Singleton

Alternativen zu Singletons

- Monostate Muster (aka borg idiom)

  - Alle Datenmitglieder der Klasse sind statisch.
  - Alle Instanzen der Klasse teilen sich die Daten.
  - Benutzer sind sich des Singleton-ähnlichen Verhaltens der Klasse nicht bewusst.

  - Lösung: `monostate.cpp`

Alternativen zu Singletons

- Dependency Injection

  - Ein Objekt injiziert die Abhängigkeit (Dienst) in ein anderes Objekt.
  - Inversion of Control → Der Injektor injiziert den Dienst das Objekt.

  - Formen der Dependency Injection
    - Konstruktor-Injektion
    - Setter-Injektion
    - Template-Parameter

  - Lösung: `dependencyInjection.cpp`

Polymorphismus

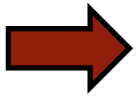
- Polymorphismus ist die Eigenschaft, dass ein Objekt ein unterschiedliches Verhalten besitzen kann.

- Dynamischer Polymorphismus
  - findet zur Laufzeit statt.
  - ist die Basis für Objekt-Orientierung.
  - ist grundlegend für Interfaces und virtuelle Methoden.
  - Benötigt eine Indirektion wie eine Referenz oder einen Zeiger.

`dispatchDynamicPolymorphism.cpp`

Polymorphismus

- Static Polymorphism
  - Polymorphismus findet zur Übersetzungszeit statt.
  - ist nicht an Schnittstellen- und Ableitungshierarchien gebunden

     Duck Typing
  - keine Indirektion wie Zeiger oder Referenzen sind notwendig
  - Statischer Polymorphismus ist typischerweise schneller als dynamischer Polymorphismus.

`dispatchStaticPolymorphism.cpp`

Polymorphismus

- Beispiele:

  - `dispatchDynamicPolymorphism.cpp`
  - `dispatchStaticPolymorphism.cpp`

- Übungen:
  - Polymorphismus kann auf verschiedene Weise umgesetzt werden. Vergleichen Sie den Polymorphismus mit `if`, mit `switch` und mit Hashtabellen mit dem bereits vorgestellten statischen und dynamischen Polymorphismus. Was sind die Vorteile der fünf Techniken?
    - Lösung: `dispatchIf.cpp`, `dispatchSwitch.cpp` und `dispatchHashtable.cpp`

CRTP

CRTP

- steht für **C**uriously **R**ecurring **T**emplate **P**attern.
- Eine Klasse wird von einem Klassen-Template abgeleitet welches sich selbst als Parameter besitzt.

```
template<class T>
class Base{
...
};

class Derived : public Base<Derived>{
...
};
```

CRTP ermöglicht statischen Polymorphismus.

templateCRTP.cpp

CRTP

Typische Anwendungsfälle

- Mixins mit CRTP
  - ermöglicht es neue Codes mit vorhandenen zu mischen.
  - Die Klasse `std::enable_shared_from_this` verwendet CRTP.
- Statischer Polymorphismus in C++.

CRTP

`std::shared_ptr` von `this` erzeugt einen `shared_pointer` für ein Objekt.

- `std::enable_shared_from_this`: Basis-Klasse für ein Objekt.
- `shared_from_this`: gibt das geteilte Objekt zurück.

```
class ShareMe: public std::enable_shared_from_this<ShareMe> {
  std::shared_ptr<ShareMe> getShared(){
    return shared_from_this();
  }
};
```

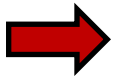
CRTP

- Beispiele:
  - `templateCRTP.cpp`
  - `templateCRTPRelational.cpp`

- Übungen:
  - Erweiteren Sie das Beispiel `templateCRTPRelation.cpp` mit einer Klasse `Person`. Eine Person sollte einen ersten und zweiten Namen haben.
    - Lösung: `templateCRTPRelational.cpp`
  - Wie können Sie in dem Beispiel `templateCRTP.cpp` vorbeugen, dass die abgeleitete Klasse einen falschen Template-Parameter hat? `Derived4: Base<Derived3>`
    - Lösung: `templateCRTPCheck.cpp`

CRTP

- Die Funktionalität des `templateCRTP.cpp` kann auf verschiedene Weise implementiert werden:
  1. Objektorientiert mit dynamischen Polymorphismus
     - Lösung: `dispatchPolymorphism.cpp`
  2. Mit einem Template
     - Lösung: `dispatchGeneric.cpp`
  3. Concepts mit C++20
     - Lösung: `dispatchConcepts.cpp`

   Implementieren Sie die erste und zweite Variante. Diskutieren Sie die Vor- und Nachteile der Varianten.
- Implementieren Sie die Klasse `ShareMe` und verwenden Sie sie.
  - Objekte der Klasse `ShareMe` sollten ein `std::shared_ptr` auf sich selbst zurückgeben.
  - Lösung: `templatesCRTPShareMe.cpp`

CRTP

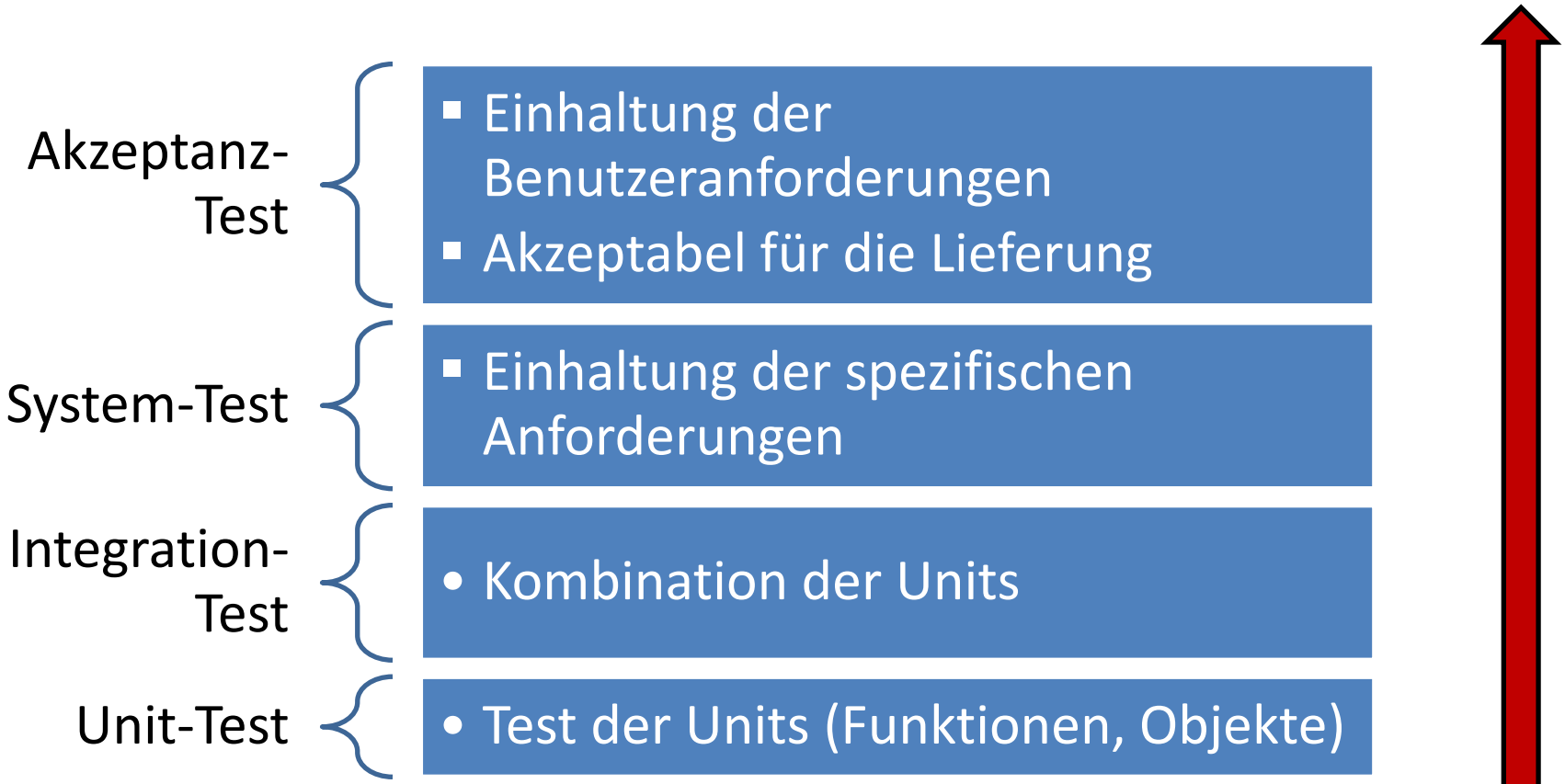
- Dank CRTP implentiert das Programm `monitorObject.cpp` das Monitor Muster. Die Klasse `ThreadSafeQueue` ist das Monitor-Objekt, das mit Hilfe der Basisklasse `Monitor` implementiert wird.

- Weitere Informationen:
  - Curiously Recurring Template Pattern
  - Fluent C++ about Templates

Clean Code in C++

- Wichtige Grundsätze
- Best Practices
- Pattern und Idiome
- **Testen**
- Refactoring
- Werkzeugunterstützung

Test Hierarchie



Funktional versus nicht-funktional

Funktionale Tests

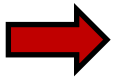
- werden vor den nicht-funktionalen Testen durchgeführt.
- überprüfen jedes Merkmal des Systems.
- basieren auf Kundenanforderungen.
- kennen nicht die Details der Software (Black-Box Tests).

- Beispiel: Kann ein Client den Server anrufen?

Nicht-funktionale Tests

- werden nach der Funktionsprüfung durchgeführt.
- verifizieren Qualitätsattribute des Systems.
- basieren auf den Kundenanforderungen.
- kennen die Details der Software (White-Box Tests).

- Beispiel: Wie viele Clients können gleichzeitig den Server kontaktieren?



Die meisten Systeme scheitern aufgrund nicht-funktionaler Anforderungen.

Charakteristik guter Unit-Tests

- **F**ast
  - schnelles Feedback (Vorbereitet für Automatisierung)
- **I**solated/**I**ndependent
  - nicht voneinander abhängig
  - haben keine externe Abhängigkeit (Datenbank)
  - Ausführung einer Zusicherung (isolierte Ausführung)
- **R**epeatable
  - wiederholbar
- **S**elf-Validating
  - keine manuelle Überprüfung der Testergebnisse wird verlangt
- **T**horough/**T**rust
  - jeden Anwendungsfall abdecken
  - robust (der Unit-Test sollte nicht scheitern)

F

I

R

S

T

Weitere Regeln (Clean C++)

- Gute Namensgebung
  - <Class to test>Test::Test<Precondition>_<Test>_<Expectation>
    `VectorTest::vectorIsEmpty_addElement_sizeIsOne`
  - <Class to Test>Test::specifyRequirements
    `AccountTest::withDrawMoneyMoreThan4000ThrowsException`
- Testen Sie nicht triviale Funktionen
  - Z.B.: getters und setters
- Code Dritter ausschließen
  - Nehmen Sie an, dass dieser Code bereits getestet wurde.

Unit-Test Patterns

Arrange **A**ct **A**ssert – AAA

- Arrange
  - Test vorbereiten
- Act
  - Test ausführen
- Assert
  - Testerfolg prüfen

Unit-Test Patterns

Test-Ausnahmen

Der Test sollte eine Ausnahme werfen.

1. Nutzen Sie eine Funktion, die die Ausnahme automatisch fängt.
2. Fangen Sie die Ausnahme im Test ab.

Unit-Test Patterns

Verschachtelte Klassen-Tests

- Protected Mitglieder einer Klasse
  - Erzeugen Sie eine abgeleitete Klasse
    1. Rufen Sie die `protected` Mitglieder von der abgeleiteten Klasse auf.
    2. Überschreiben Sie virtuelle Funktionen und rufen Sie sie auf.

Unit-Test Patterns

Interaktions Test

- Erzeugen Sie ein Fake-Objekt (test doubles, mock objects).
  - Das reale Objekt
    - ist nicht deterministisch.
    - ist schwierig einzurichten.
    - ist langsam.
    - hat ein Benutzerinerface.
    - existiert noch nicht.

Unit-Test Muster

Verwendung eines Mock Frameworks

- Frameworks mit Mocks (Wikipedia)
  - Boost test library (with Turtle)
  - CppUTest
  - CUTE
  - Google Test (mith googlemock)
- Separate Mocks (Wikipedia)
  - FakeIt
  - Hippo Mocks
  - Mockator
  - Trompeloeil

Unit-Test Patterns

- Beispiele:
  - `unitTest.cpp`

Übungen:

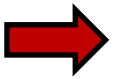
- Die Datei `unitTest.cpp` zeigt Beispiele der ersten drei Muster. Es nutzt das Google Test Framework. Studieren Sie das Beispiel.
- Erweitern Sie den Test in `unitTest.cpp` Datei.
  - Exakte Übereinstimmung bei Fließkommazahlen ist ein Antipattern. Verbessern Sie das Programm, indem Sie aus der exakten Übereinstimmung eine Übereinstimmung mit Fehlertoleranz umsetzen.
  - Fangen Sie die Ausnahme direkt im Test ab.
  - Lösung: `unitTestExcercise.cpp`
- Warum ist die Verwendung von `protected` Mitglieder einer Klassenhierarchie schlecht?

Antipattern für Unit-Tests

- Singletons (statische Mitglieder einer Klasse)
- Nicht-virtuelle Funktionen einer Klasse
- Globale Objekte

Einführung von Unit-Tests

- Hinzufügen von Unit-Tests zum Legacy-Code, falls
  - neue Funktionalität hinzugefügt wird.
  - ein Bug gefunden wurde.
  - schwieriger Code dokumentiert werden muss.
  - eine Refaktorierung des Codes bevorsteht.

 Boy Scout Rule

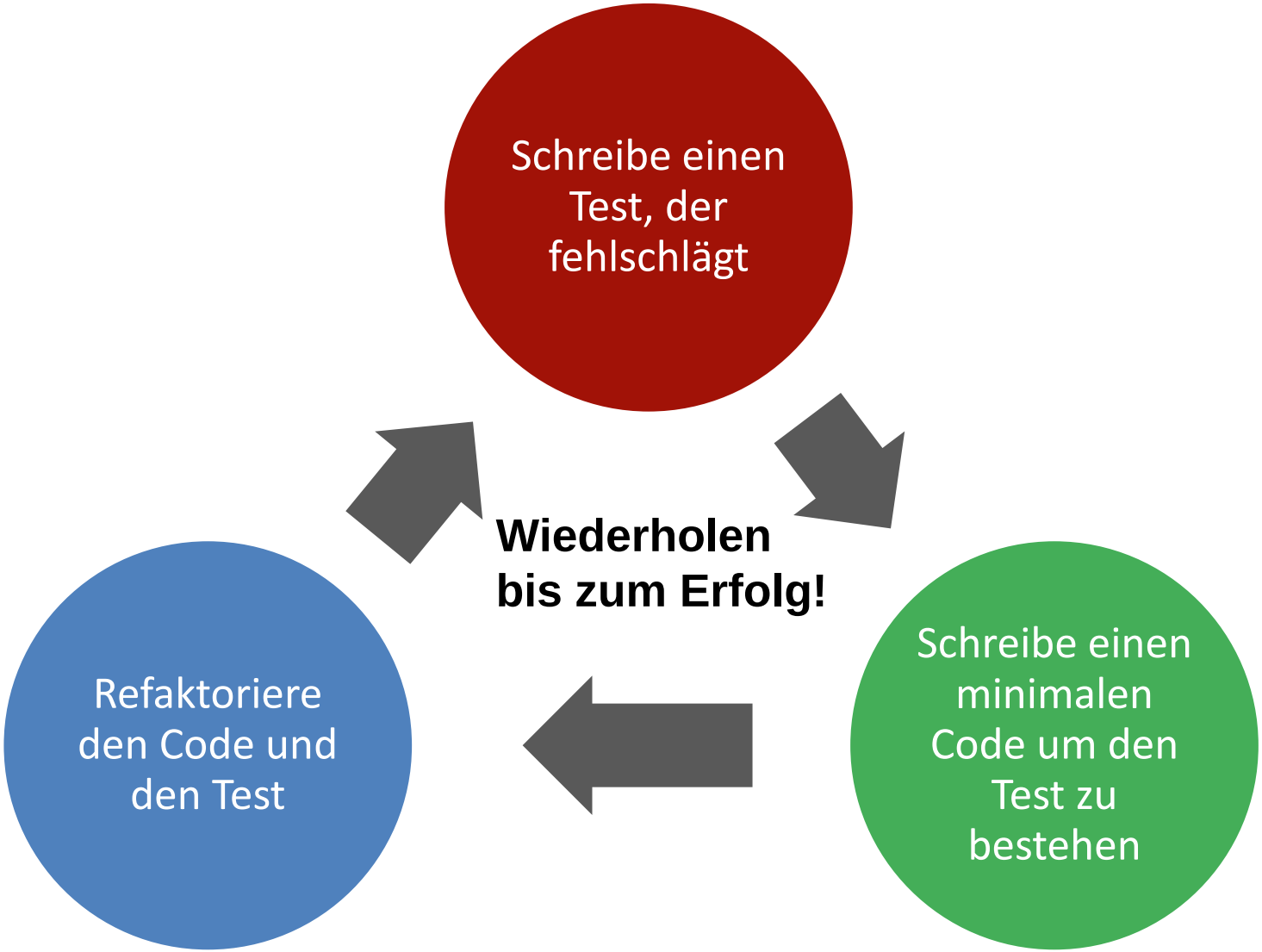
Test-Driven Entwicklung

Nachteile der **P**lain **O**ld **U**nit-**T**ests (POUT)

- Kein Zwang die Tests zu implementieren.
- Der resultierende Code ist schwer zu testen.
- Schwierigkeiten, eine hohe Codeabdeckung zu erreichen.

⇨ Schreiben Sie den Test zuerst.

Test-Driven Entwicklung



TDD – Vorteile/Nachteile

- Vorteile
  - zwingt zu kleinen Schritt
  - schnelle Feedback-Schleifen
  - zwingt zu Reflektieren, was getan werden muss
  - erstellt eine ausführbare Spezifikation
  - bewusster Umgang mit Abhängigkeiten
  - “You know when your are done”!
  - 100 % Testabdeckung per Definition

- Aber nutzen sie nicht TDD für
  - einfachen Code
  - Prototyping

Test-Driven Entwicklung

- Beispiele:
  - `ArabicToRomanTDD`

- Übungen:
  - Verwenden Sie die test-driven Entwicklung, um eine römisch-arabische Conversion-Funktion zu implementieren.
  - Lösung: `RomanToArabicTDD`

Clean Code in C++

- Wichtige Grundsätze
- Best Practices
- Pattern und Idiome
- Testen
- **Refactoring**
- Werkzeugunterstützung

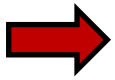
Zentrale Ideen

Code Refaktoring

- Neustrukturierung des bestehenden Codes ohne Änderung des externen Verhaltens.

- Vorteile
  - Wartbarkeit
  - Erweiterbarkeit

- Sicherheitsnetz
  - Tests
  - IDE Support

 Refactoring wird durch Code-Smells motiviert.

Code Smells

Code Smell

- Verletzung grundlegender Designprinzipien, die sich auf die Designqualität auswirken

- Anwendungsebene
  - duplizierter Code
  - zu komplizierter Code
  - eine Änderung betrifft mehrere Klassen
  - globaler gemeinsamer Zustand
  - magische Zahlen
  - magische Namen

Code Smells

- Klassen-Level
  - große Klasse; Minimale Klasse
  - starke Kopplung von Klassen
  - das Methoden-Überschreiben bricht den Vertrag der Basisklasse
  - downcasting
- Funktions-Level
  - zu viele Parameter
  - lange Funktion
  - lange Codezeilen
  - redundante Kommentare
  - tiefe Kontrollstrukturen

Ziele

- Besseres Verständnis
  - Abhängigkeiten entdecken
- Mehr Abstraktion
  - bessere Kapselung
  - Generalisierung (Templates)
  - Ersetzen von Bedingungen
- Trennung der Zuständigkeiten
  - Komponentenbildung
  - Klassen oder Funktionen extrahieren
- Verbesserung von Namen und der Struktur des Codes
  - Funktionen oder Variablen umbenennen
  - Verschieben von Funktion in Objekthierachien
- Erkennen von Duplikaten

Taktiken

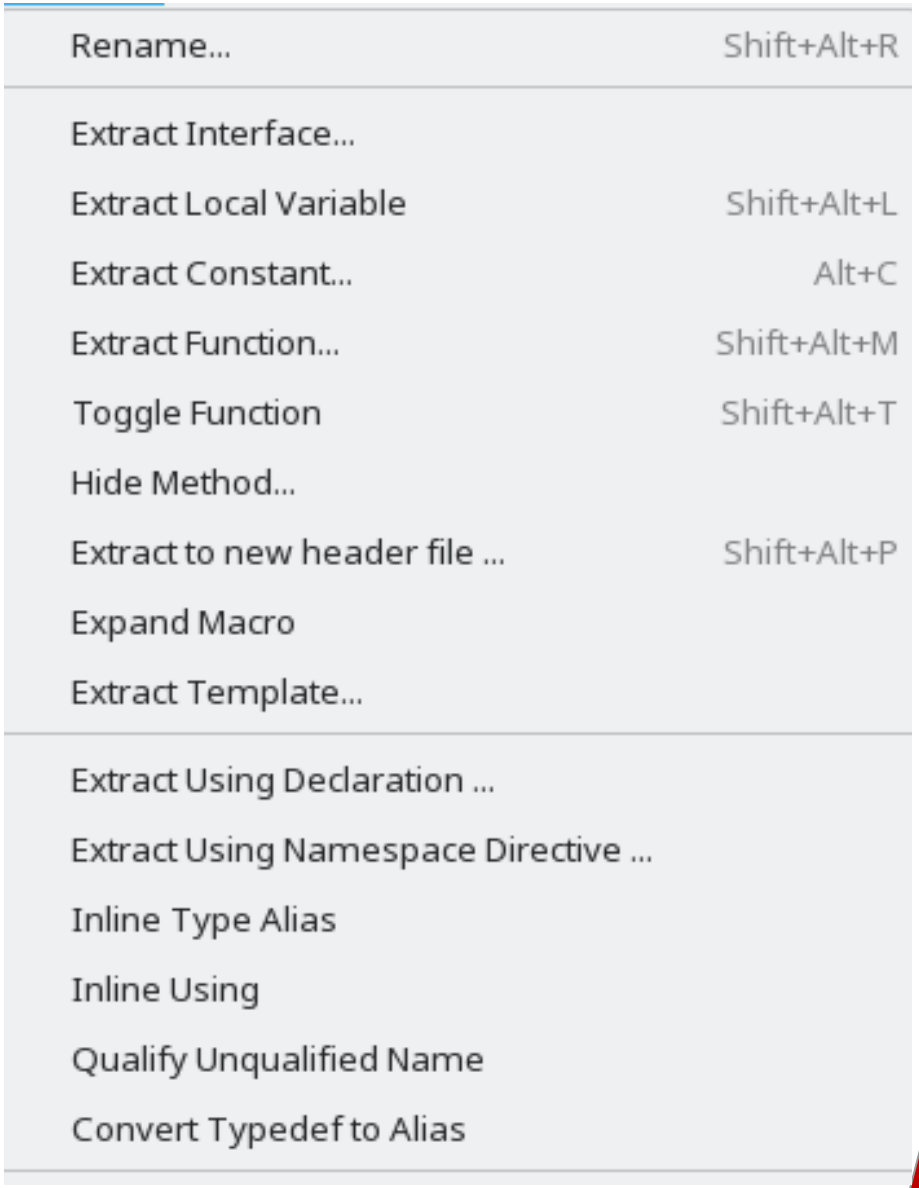
Planen Sie große Refaktorierungen

- Das Ziel einschränken
  - Machen Sie sich Notizen über zusätzliche Code-Smells.
- Beschränken Sie die Zeit
- Die Codestellen einschränken
  - Finden Sie die Hot Spots in Ihrer Anwendung.
- Kleine Schritte machen
  - Überprüfen Sie jeden Schritt.
    - Wenn Sie keine Unit-Tests haben, erstellen Sie zumindest System-Tests.
  - Streben Sie schnelles Feedback an
    - Refaktoring sollte ein frühes Feedback geben, damit Sie wissen, ob Sie auf dem richtigen Pfad sind.

Werkzeugunterstützung

- Eclipse
  - CDT (C/C++ Entwicklung tooling)
  - Cevelop
- Visual Studio Code (Umbenennung)

- Commercial
  - Qt Creator
  - Clion
  - ReSharper (Visual Studio Erweiterung)



Techniken (Cevelop)

- `Rename`
  - Umbenennung von Variablen, Funktionen und Klassen
- `Extract Interface...`
  - erstellt eine neue Headerdatei
  - Wählen Sie aus, welche Funktion rein virtuell werden soll.
- `Extract Local Varaiable`
  - Eine neue Variable für den Ausdruck einführen.
- `Extract Constant`
  - macht aus einem Literal eine Konstante
- `Exract Function`
  - extrahiert eine Funktion aus ein paar Codezeilen

Techniken (Cevelop)

- `Toggle Function`
  - verschiebt eine Funktion aus einer Sourcecodedatei in eine Headerdatei und umgekehrt
- `Hide Method…`
  - macht eine Funktion `private`
- `Extract to new header file…`
  - extrahiert eine Funktion oder eine Klasse in eine neue Kopfzeile
  - fügt die neue Kopfzeile automatisch ein
- `Expand Macro`
  - ersetzt Makros mit `constexpr` Variablen oder Funktionen
- `Extract Template…`
  - macht ein Template aus einer Funktion oder einer Klasse

Techniken (Cevelop)

- `Extract Using Declaration…`
  - extrahiert eine Using Declaration: `using std::vector;`
- `Extract Using Namespace Directive…`
  - extrahiert eine Using Directive: `using namespace std;`
- `Inling Using`
  - das Gegenteil von der Extract Using
- `Inline Type Alias`
  - ersetzt Aliase: `using Vector = std::vector;`
- `Qualifizy Unqualified Name`
  - macht die Namen qualifiziert: `std::vector`
- `Konvertiert Typedef to Alias`
  - konvertiert zur Alias Syntax: `using sum = int (+) (int, int)`

Refactoring

- Beispiele:

  - `refactor1.cpp, refactor2.cpp, refactor3.cpp`
  - `dispatchIf.cpp, dispatchSwitch.cpp, dispatchHashtable.cpp, dispatchDynamicPolymorphism.cpp, dispatchStaticPolymorphism.cpp`
- Übungen:
  - Studieren Sie die Programme `refactor1.cpp`, `refactor2.cpp` und `refactor3.cpp`.
    - Analysieren Sie das Programmverhalten.
    - Verifizieren Sie ihre Vermutung durch das Ausführen der Programme.
    - Refaktorieren Sie die Programme.
    - Lösung: `refactor1.cpp`, `refactor2.cpp` und `refactor3.cpp`.

Refactoring

- Bedingtes ausführen mit `if/else` (`dispatchIf.cpp`) oder `switch`(`dispatchSwitch.cpp`) zu ersetzen ist eine häufig angewandte Refaktorierungs-Technik. Hashtables (`dispatchHashtable.cpp`), dynamic polymorphism (`dispatchDynamicPolymorphism.cpp`) und static polymorphism (`dispatchStaticPolymorphism.cpp`) sind weitere Techniken für bedingtes ausführen. Wann würden Sie welche Technik bevorzugen?

Clean Code in C++

Wichtige Grundsätze

Best Practices

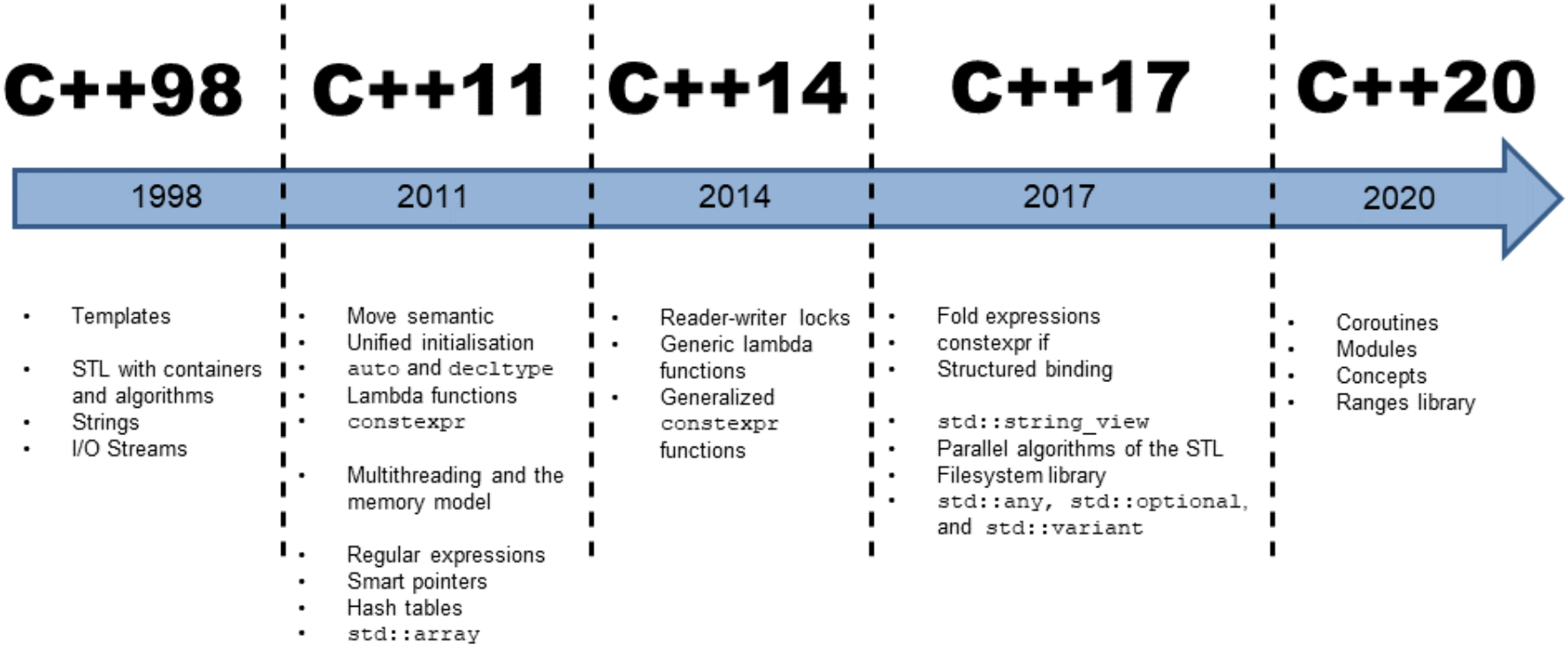
Pattern und Idiome

Testen

Refactoring

Werkzeugunterstützung

Die C++ Standards



Die Großen Drei – Die Details

Unterstützung der C++ Standards: cppreference.com

C++14 feature	Paper(s)	Version	GCC	Clang	MSVC	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)
Tweaked wording for contextual conversions	N3323	c++14-lang	4.9	3.4	18.0*	4.9	16.0	13.1.2*	5.15			
Binary literals	N3472	c++14-lang	4.3/4.9	2.9	19.0*	4.10	11.0	13.1.2*	5.14			
decltype(auto), Return type deduction for normal functions	N3638	c++14-lang	4.8/4.9	3.3/3.4	19.0*	4.9	15.0	13.1.2*	5.15			
Initialized/Generalized lambda captures (init-capture)	N3648	c++14-lang	4.5/4.9	3.4	19.0*	4.10	15.0		5.15			
Generic (polymorphic) lambda expressions	N3649	c++14-lang	4.9	3.4	19.0*	4.10	16.0	13.1.2*	5.15			
Variable templates	N3651	c++14-lang	5	3.4	19.0*	4.11	17.0	13.1.2*	5.15			
Extended constexpr	N3652	c++14-lang	5	3.4	19.10*	4.11	17.0	13.1.2*	5.15			
Member initializers and aggregates (NSDMI)	N3653	c++14-lang	5	3.3	19.10*	4.9	16.0		5.14			
Clarifying memory allocation (avoiding/fusing allocations)	N3664	c++14-lang	N/A	3.4	N/A	N/A			N/A			
[[deprecated]] attribute	N3760	c++14-lang	4.9	3.4	19.0*	4.9	15.0* 16.0	13.1.2*	5.14			
Sized deallocation	N3778	c++14-lang	5	3.4	19.0*	4.10.1	17.0		5.14			

C++11 feature	Paper(s)	Version	GCC	Clang	MSVC	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)	HP aCC	Digital Mars C++
alignas	N2341	c++11	4.8	3.0	19.0*	4.8	15.0	13.1.2*	5.13	Yes	8.6	2015		
alignof	N2341	c++11	4.5	2.9	19.0*	4.8	15.0	13.1.2*	5.13	Yes	8.4	2015		
Atomic operations	N2427	c++11	4.4	3.1	17.0*	Yes	13.0	13.1.2*	5.14	Yes	8.4	2015		
auto	N1984(v1.0)	c++11	4.4(v1.0)	Yes	16.0*	4.1(v0.9)	11.0(v0.9) 12.0(v1.0)	11.1(v1.0)	5.13	Yes	8.4	2015	A.06.25	
C99 preprocessor	N1653	c++11	4.3	Yes	19.0* (partial - buggy variadic macros)	4.1	11.1	10.1	5.9	Yes	8.4	2015	A.06.25	Yes
constexpr	N2235	c++11	4.6	3.1	19.0* (partial)	4.6	13.0* 14.0	12.1* 13.1	5.13	Yes	8.4	2015	A.06.28	
decltype	v1.0: N2343 v1.1: N3276	c++11	4.3(v1.0) 4.8.1(v1.1)	2.9	16.0*	4.1(v1.0)	11.0(v1.0) 12.0(v1.1)	11.1(v1.0)	5.13	Yes	8.4	2015	A.06.25	8.52(v1.0)
Defaulted and deleted functions	N2346	c++11	4.4	3.0	18.0*	4.1	12.0	13.1	5.13	Yes	8.4	2015	A.06.25	
Delegating constructors	N1986	c++11	4.7	3.0	18.0*	4.7	14.0	11.1	5.13	Yes	8.4	2015	A.06.28	
Explicit conversion operators	N2437	c++11	4.5	3.0	18.0*	4.4	13.0	12.1	5.13	Yes	8.4	2015	A.06.27	
Extended `friend` declarations	N1791	c++11	4.7	2.9	16.0* (partial)	4.1	11.1* 12.0	11.1	5.13	Yes	8.4	2015	A.06.25	
`extern template`	N1987	c++11	3.3	Yes	12.0*	3.9	9.0	11.1	5.13	Yes	8.4	2015	A.06.25	
Forward `enum` declarations	N2764	c++11	4.6	3.1	17.0*	4.5	11.1* 14.0	12.1	5.13	Yes	8.4	2015		

Die großen Drei – Die Details

C++2a feature	Paper(s)	Version	GCC	Clang	MSVC	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)
Allow lambda-capture [=, this]	P0409R2	c++2a-lang	8	6								
__VA_OPT__	P0306R4	c++2a-lang	8 (partial)*	6								
Designated initializers	P0329R4	c++2a-lang	4.7 (partial)* 8	3.0 (partial)*								
template-parameter-list for generic lambdas	P0428R2	c++2a-lang	8									
Default member initializers for bit-fields	P0683R1	c++2a-lang	8	6								
Initializer list constructors in class template argument deduction	P0702R1	c++2a-lang	8	6								
const&-qualified pointers to members	P0704R1	c++2a-lang	8	6								
Concepts	P0734R0	c++2a-lang	6 (TS only)									
Lambdas in unevaluated contexts	P0315R4	c++2a-lang	9									
Three-way comparison operator	P0515R3	c++2a-lang		8 (partial)*								

C++17 feature	Paper(s)	Version	GCC	Clang	MSVC	EDG eccp	Intel C++	IBM XLC++	Sun/Oracle C++	Embarcadero C++ Builder	Cray	Portland Group (PGI)
New auto rules for direct-list-initialization	N3922	c++17-lang	5	3.8	19.0*	4.10.1	17.0					17.7
static_assert with no message	N3928	c++17-lang	6	2.5	19.10*	4.12	18.0					17.7
typename in a template template parameter	N4051	c++17-lang	5	3.5	19.0*	4.10.1	17.0					17.7
Removing trigraphs	N4086	c++17-lang	5	3.5	16.0*	5.0						
Nested namespace definition	N4230	c++17-lang	6	3.6	19.0*	4.12	17.0					17.7
Attributes for namespaces and enumerators	N4266	c++17-lang	4.9 (namespaces) / 6 (enumerators)	3.6	19.0*	4.11	17.0					17.7
u8 character literals	N4267	c++17-lang	6	3.6	19.0*	4.11	17.0					17.7
Allow constant evaluation for all non-type template arguments	N4268	c++17-lang	6	3.6	19.12*	5.0						
Fold Expressions	N4295	c++17-lang	6	3.6	19.12*	4.14	19.0					18.1
Remove Deprecated Use of the register Keyword	P0001R1	c++17-lang	7	3.8	19.11*	4.13	18.0					17.7

Sanitizer

- Sanitizer
  - GCC >= 4.8
  - Clang >= 3.2
  - MSVC (Visual Studio >= 16.4)

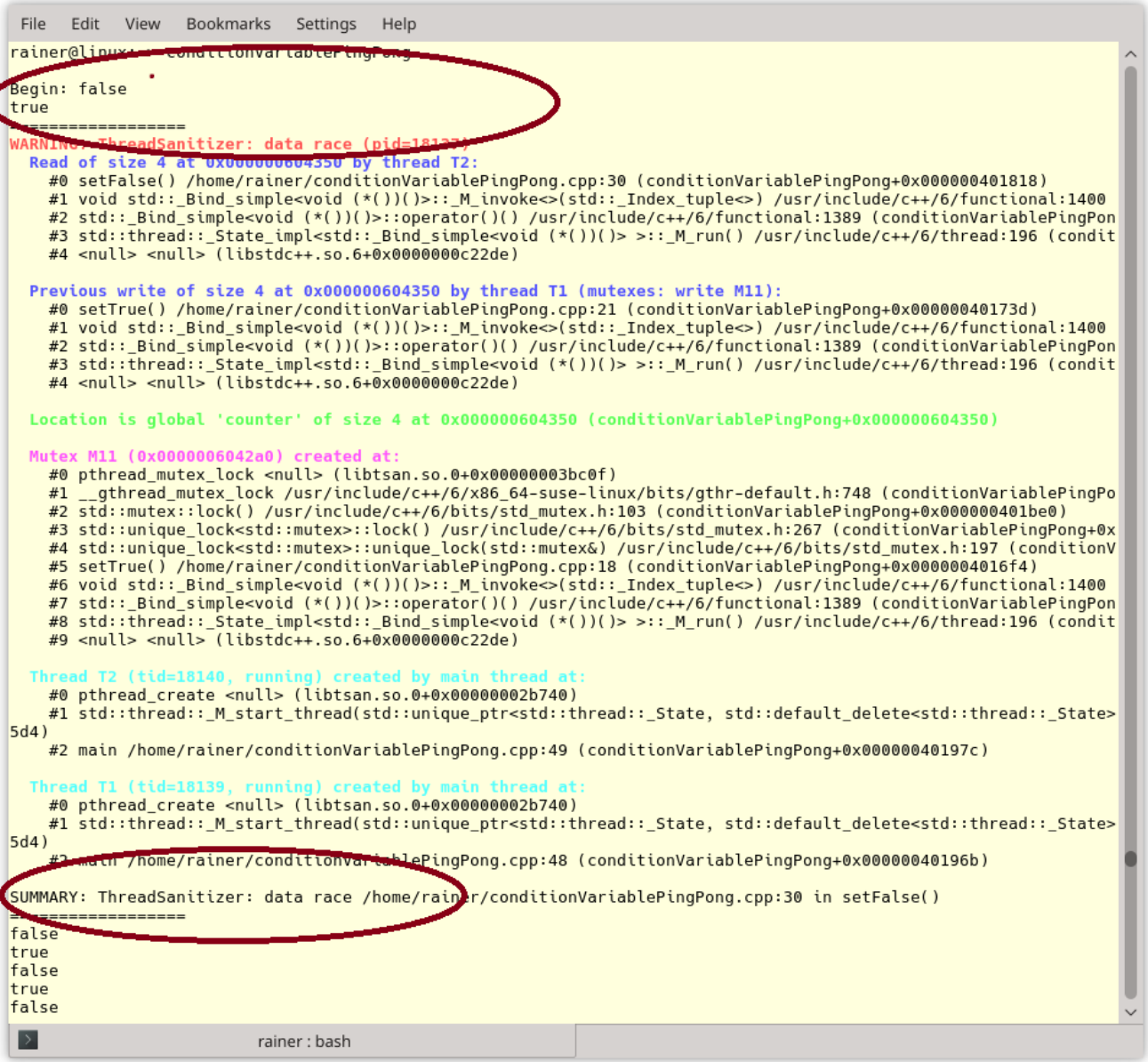
- Dynamic Code Analysis
  - AddressSanitizer (Adressen): `-fsanitize=address -g`
  - LeakSanitizer (Speicherlöcher): `-fsanitize=address -g`
  - ThreadSanitizer (Data Races und Deadlocks): `-fsanitize=thread -g`
  - MemorySanitizer (Uninitialisierter Speicher): `-fsanitize=memory -g`

ThreadSanitizer

```cpp
bool dataReady= false;

std::mutex mutex_;
std::condition_variable condVar1;
std::condition_variable condVar2;

int counter=0;
int COUNTLIMIT=50;

void setTrue(){

  while(counter <= COUNTLIMIT){

    std::unique_lock<std::mutex> lck(mutex_);
    condVar1.wait(lck,[]{return dataReady == false;});
    dataReady= true;
    ++counter;
    std::cout << dataReady << std::endl;
    condVar2.notify_one();

  }
}
```

```cpp
void setFalse(){

  while(counter < COUNTLIMIT){

    std::unique_lock<std::mutex> lck(mutex_);
    condVar2.wait(lck,[]{return dataReady == true;});
    dataReady= false;
    std::cout << dataReady << std::endl;
    condVar1.notify_one();

  }

}

int main(){

  std::cout << std::boolalpha << std::endl;

  std::cout << "Begin: " << dataReady << std::endl;

  std::thread t1(setTrue);
  std::thread t2(setFalse);

  t1.join();
  t2.join();

  dataReady= false;
  std::cout << "End: " << dataReady << std::endl;

  std::cout << std::endl;
}
```

ThreadSanitizer



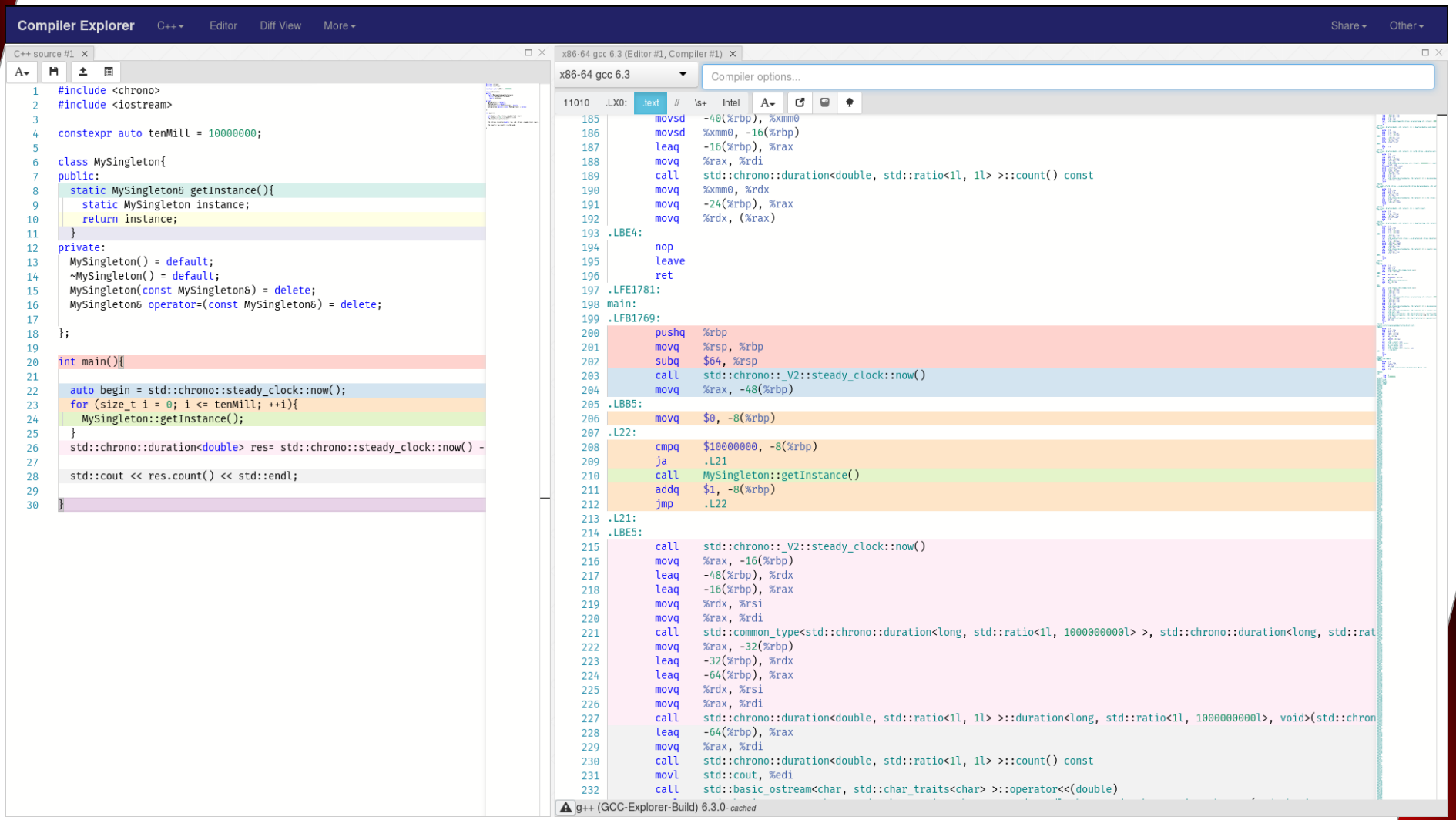
Compiler Explorer

Compiler Explorer (Matt Godbolt) erzeugt Assembler-Anweisungen.

- unterstützt mehr als 60 Compiler
- stellt Sourcecode den Assembler-Instruktionen gegenüber
- führt den Code aus
- unterstützt mehrere Compiler gleichzeitig

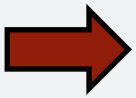
- Die Vorteile:
  - gibt tiefe Einblicke in den Übersetzung- und Optimierungsprozess
  - zeigt an, ob Code zur Compilezeit oder Laufzeit ausgeführt wird
  - zeigt an, ob eine Funktion `inline` ausgeführt wird.
  - stellt die Assembler-Instruktionen in Abhängigkeit von den Compiler-Flags dar

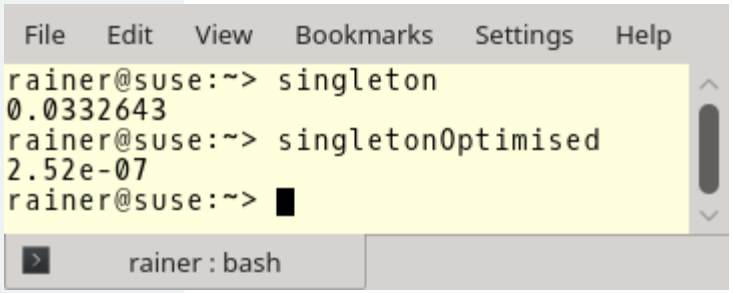
Compiler Explorer



Compiler Explorer

```cpp
#include <chrono>
#include <iostream>

constexpr auto tenMill = 10000000;

class MySingleton{
public:
  static MySingleton& getInstance(){
    static MySingleton instance;
    return instance;
  }
private:
  MySingleton() = default;
  ~MySingleton() = default;
  MySingleton(const MySingleton&) = delete;
  MySingleton& operator=(const MySingleton&) = delete;

};

int main(){

  auto begin = std::chrono::steady_clock::now();
  for (size_t i = 0; i <= tenMill; ++i){
    MySingleton::getInstance();
  }
  std::chrono::duration<double> res= std::chrono::steady_clock::now() - begin;

  std::cout << res.count() << std::endl;

}
```





Compiler Explorer

Nicht optimierte Variante

```
19
20  int main(){
21
22    auto begin = std::chrono::steady_clock::now();
23    for (size_t i = 0; i <= tenMill; ++i){
24      MySingleton::getInstance();
25    }
26    std::chrono::duration<double> res= std::chrono::steady_clock::now() -
27
28    std::cout << res.count() << std::endl;
29
30  }
```

```
207 .L22:
208         cmpq    $10000000, -8(%rbp)
209         ja      .L21
210         call    MySingleton::getInstance()
211         addq    $1, -8(%rbp)
212         jmp     .L22
213 .L21:
```

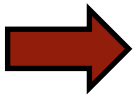
Optimierte Variante

```
20  int main(){
21
22    auto begin = std::chrono::steady_clock::now();
23    for (size_t i = 0; i <= tenMill; ++i){
24      MySingleton::getInstance();
25    }
26    std::chrono::duration<double> res= std::chrono::steady_clock::now() -
27
28    std::cout << res.count() << std::endl;
29
30  }
```

```
2 main:
3 .LFB1793:
4         pushq   %rbx
5         call    std::chrono::_V2::steady_clock::now()
6 .LVL0:
7         movq    %rax, %rbx
8 .LVL1:
9         call    std::chrono::_V2::steady_clock::now()
```

Compiler Explorer

```cpp
#include <iostream>

constexpr int gcd(int a, int b){
  while (b != 0){
    auto t = b;
    b = a % b;
    a = t;
  }
  return a;
}

int main(){

  auto res = gcd(121, 11);
  constexpr auto res1 = gcd(100, 10);

  std::cout << "res: " << res << std::endl;
  std::cout << "res1: " << res1 << std::endl;

}
```



```
main:
  push rbp
  mov rbp, rsp
  sub rsp, 16
  mov esi, 11
  mov edi, 121
  call gcd(int, int)
  mov DWORD PTR [rbp-4], eax
  mov DWORD PTR [rbp-8], 10
  mov esi, OFFSET FLAT:.LC0
  mov edi, OFFSET FLAT:_ZSt4cout
  call std::basic_ostream<char, std::char_tra
```

Compiler Explorer

- Beispiele
  - gcd
  - Singleton

C++ Insights

C++ Insights (Andreas Fertig) enthüllt die Compiler Magie.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main(){

    std::cout << std::endl;

    std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
    std::transform(vec.begin(), vec.end(), vec.begin(),
                   [](auto i){ return i * i; });

    for (auto v: vec){
        std::cout << v << " ";
    }

    std::cout << "\n\n";

}
```

Insight:

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

int main(){

    std::cout.operator<<(std::endl);

    std::vector<int> vec{ std::initializer_list<int>{1, 2, 3, 4, 5, 6, 7, 8, 9} };

     class __lambda_10_56
     {
       public: inline /*constexpr */ int operator()(int i) const
       {
         return i * i;
       }

     };

     std::transform(vec.begin(), vec.end(), vec.begin(), __lambda_10_56{});

    {
      std::vector<int, std::allocator<int> > & __range = vec;
      __gnu_cxx::__normal_iterator<int *, std::vector<int, std::allocator<int> > > __begin = __range.begin();
      __gnu_cxx::__normal_iterator<int *, std::vector<int, std::allocator<int> > > __end = __range.end();

      for( ; __gnu_cxx::operator!=(__begin, __end); __begin.operator++() )
      {
        int v = __begin.operator*();
        std::operator<<(std::cout.operator<<(v), " ");
      }
    }

    std::operator<<(std::cout, "\n\n");

}
```

C++ Insights

```cpp
#include <utility>

struct MyType{
    MyType(int, double, bool){};
};

template <typename T, typename ... Args>
T createT(Args&& ... args){
    return T(std::forward<Args>(args) ... );
}

int main(){

    int lvalue{2020};

    int uniqZero = createT<int>();                      // (1)
    auto uniqEleven = createT<int>(2011);               // (2)
    auto uniqTwenty = createT<int>(lvalue);             // (3)
    auto uniqType = createT<MyType>(lvalue, 3.14, true); // (4)

}
```

```cpp
/* First instantiated from: insights.cpp:18 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int createT<int, >()
{
  return int();
}
#endif


/* First instantiated from: insights.cpp:19 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int createT<int, int>(int && __args0)
{
  return int(std::forward<int>(__args0));
}
#endif
```

```cpp
/* First instantiated from: insights.cpp:20 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
int createT<int, int &>(int & args)
{
  return int(std::forward<int &>(args));
}
#endif


/* First instantiated from: insights.cpp:21 */
#ifdef INSIGHTS_USE_TEMPLATE
template<>
MyType createT<MyType, int &, double, bool>(int & args, double && __args1, bool && __args2)
{
  return MyType(std::forward<int &>(args), std::forward<double>(__args1), std::forward<bool>(__args2));
}
#endif
```

C++ Insights

- Beispiele
  - auto
  - Automatisch erzeugte Methoden
  - Vorzeichenlose und vorzeichenbehaftete Arithmetik vermischen
  - Range-basierte für-Schleife
  - Lambda-Funktionen
  - Perfect-fürwarding mit Variadic Templates
  - Operatoren Überladung
  - Templates

Guidelines Support Library

Die Guidelines Support Library (GSL) ist eine kleine Bibliothek zur Unterstützung der Guidelines der C++ Core Guidelines.

- Die GSL
  - ist in Headerdateien implementiert.
  - besitzt viele Implementierungen.
    - Microsoft/GSL
    - GSL-Lite basiert auf C++98
  - antizipiert häufig den C++ Standard.
- Die Komponenten
  - Views (views sind niemals Besitzer)
  - Besitzer
  - Assertions (Contracts, die auf Makros basieren.)
  - Utilities (Neue, sichere Casts)

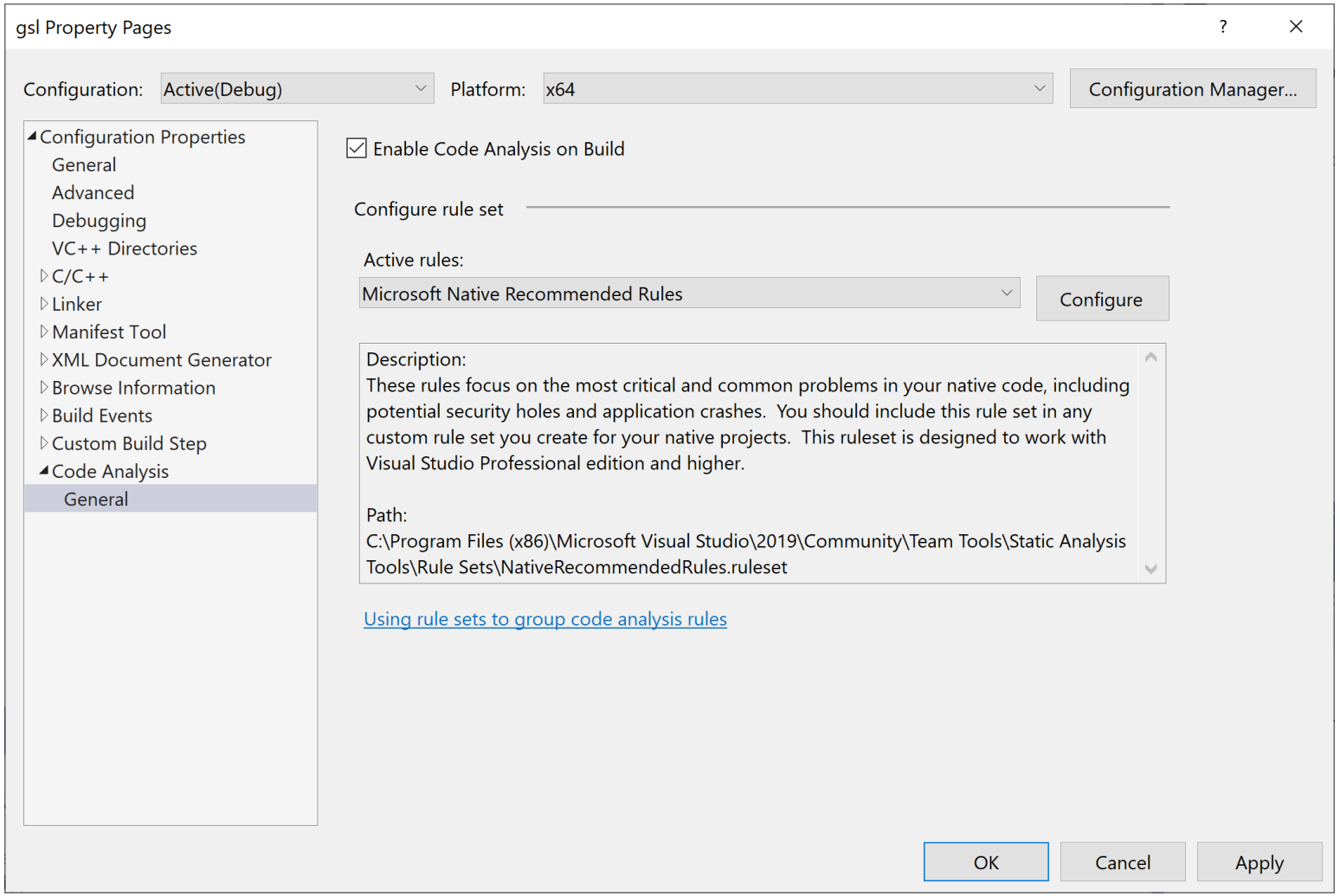
Guidelines Support Library

Bruch der Type-Safety, Bounds-Safety und Lifetime-Safety.

```
#include <iostream>

void f(int* p, int count) {}
void f2(int* p) { int x = *p; }

int main() {
    // Break of type safety
    double d = 2;
    auto p = (long*)&d;
    auto q = (long long*)&d;
    // Break of bounds safety
    int myArray[100];
    f(myArray, 100);
    // Break of lifetime safety
    int* a = new int;
    delete a;
    f2(a);
}
```

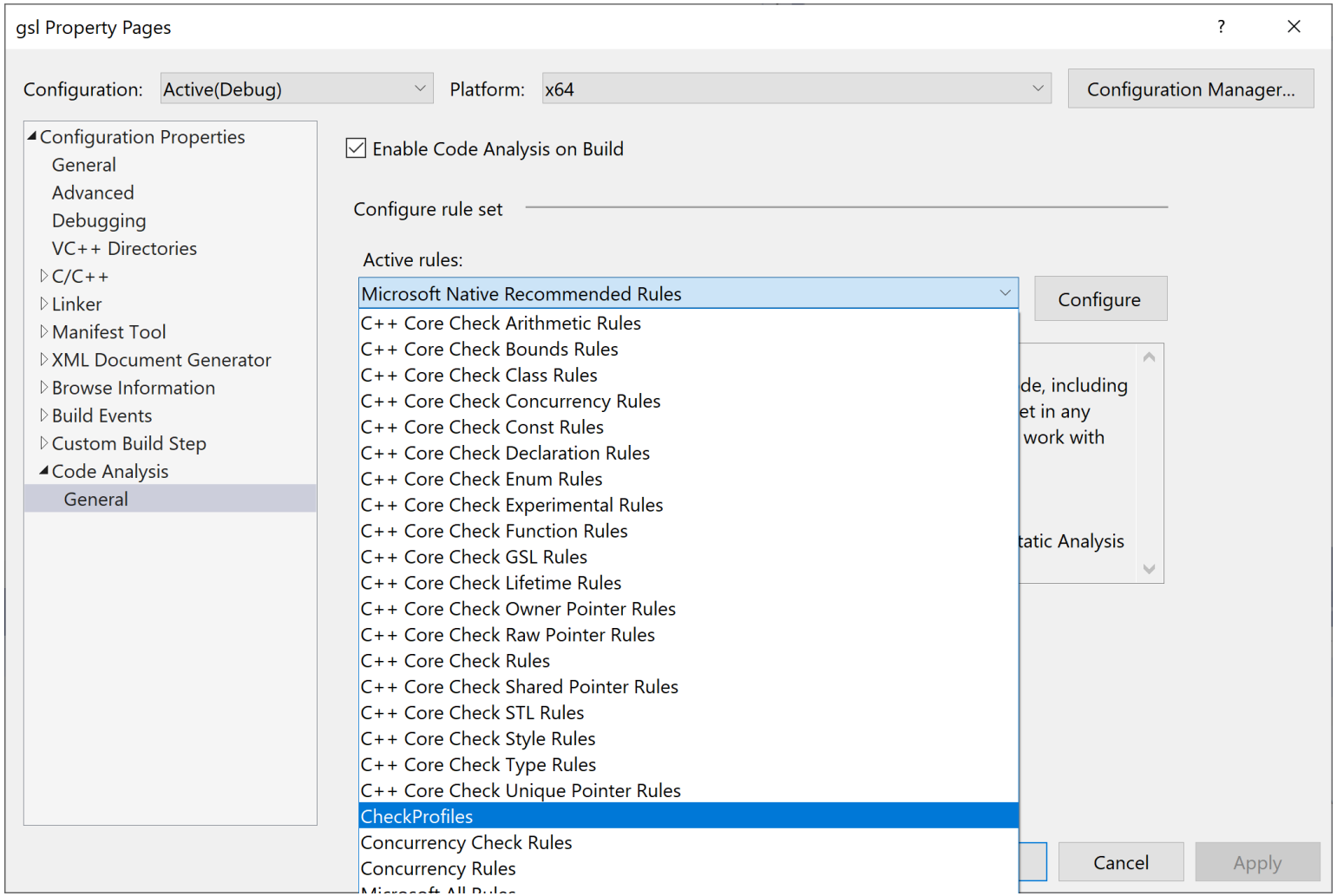
Guidelines Support Library

Aktivieren der Code-Analyse beim Build



Guidelines Support Library

Konfigurieren der verwendeten Regeln

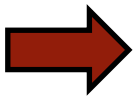


Guidelines Support Library

Code-Analyse der Lösung ausführen

```
1>------ Rebuild All started: Project: gsl, Configuration: Debug x64 ------
1>gsl.cpp
1>gsl.vcxproj -> C:\Users\raine\source\repos\gsl\x64\Debug\gsl.exe
C:\Users\raine\source\repos\gsl\gsl\gsl.cpp(17): warning C26493: Don't use C-style casts (type.4).
C:\Users\raine\source\repos\gsl\gsl\gsl.cpp(18): warning C26493: Don't use C-style casts (type.4).
C:\Users\raine\source\repos\gsl\gsl\gsl.cpp(29): warning C26486: Don't pass a pointer that may be invalid to a function. Parameter 0 'a' in call to 'f2' may be invalid (lifetime.3).
C:\Users\raine\source\repos\gsl\gsl\gsl.cpp(24): warning C26485: Expression 'myArray': No array to pointer decay (bounds.3).
1>Done building project "gsl.vcxproj".
========== Rebuild All: 1 succeeded, 0 failed, 0 skipped ==========
```

Warnungen unterdrücken



```cpp
#include <iostream>

void f(int* p, int count) {}

int main() {
    int myArray[100];
    // Break of bounds safety
    [[gsl::suppress(bounds.3)]] {    // suppress warning
        f(myArray, 100);
    }
    f(myArray, 100);                 // warning
}
```