IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz Universität Hannover

Programmierpraktikum Technische Informatik (C++)

5.5.2022

IMS
Institut für Mikroelektronische Systeme
Leibniz Universität Hannover

Leibniz
Universität
Hannover

Objektorientierung in C++

Objektorientierte Programmierung

- Unterstützung für objektorientierte Programmierung (OOP) ist wichtigste Neuerung von C++
  - Ursprünglicher Name für C++: **C with Classes**
- OOP aus Java bereits bekannt
- Kurze Wiederholung der wichtigsten Konzepte zur Auffrischung

Auffrischung OOP

- Zusammenfassung von Daten und Funktionen in Typen (Klassen)
  - Betrachtung von Daten und assoziierten Funktionen als Einheit erhöht Wartbarkeit
  - Bündelung zusammengehöriger Daten kann durch Cacheeffekte Performancevorteile bringen
- Objekte sind Instanzen von Klassen
- Objekte derselben Klasse besitzen denselben Aufbau
  - Können aber unterschiedliche Datenwerte enthalten

Konzepte der OOP

- Abstraktion
  - Definition klar ersichtlicher Interfaces zur Manipulation von Daten
  - Trennung von Interface und Implementation
- Kapselung
  - Zugriff auf Daten einer Klasse nur über öffentliche Interfaces
  - Zugriff auf Interna nicht von außen möglich
  - Garantiert die Einhaltung von Invarianten

Konzepte der OOP II

- Vererbung
  - Klassen können Eigenschaften von anderen Klassen erben
  - Implementation gemeinsamer Eigenschaften in einer Basisklasse
  - Verringert Codeduplikation
  - Vererbung ist eine **is-a**-Beziehung
  - Instanzen einer abgeleiteten Klasse als Instanzen der Basisklasse verwendbar
- Polymorphie
  - Funktionen aus unterschiedlichen Klassen können über identisches Interface angesprochen werden
- Implementation von Vererbung und Polymorphie in C++ erst in den nächsten Veranstaltungen

Ein paar Worte zum Codingstyle

- Objektorientierte Programmierung in C++ erzeugt viele Scopes
- In der Vorlesung verwendeter Klammersetzungsstil führt zu vielen fast leeren Zeilen
- Folien bieten nur einen begrenzten Platz
- Abweichungen vom eigentlichen Stil in den kommenden Folien
- Sichtbarkeiten von Membern auf einigen Folien nur aus Platzgründen nicht immer spezifiziert

In echtem Sourcecode einheitlichen Programmierstil verwenden und Sichtbarkeit für Member explizit angeben!

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- Zwei Arten objektorientierter Datentypen: `class` und `struct`
- Im Gegensatz zu C wird durch `class Interval` oder `struct Interval` direkt ein Typ `Interval` definiert
  - Kein expliziter `typedef` notwendig

Klassendeklaration und -definition

- Klassen werden mit `class` *name*`;` (forward) deklariert
- Rumpf einer Klasse steht in geschweiften Klammern
- Klassenrumpf bildet einen Scope, enthaltene Variablen/Funktionen können nur über die Klasse angesprochen werden
- Definition und Deklaration werden jeweils mit `;` abgeschlossen
  - Kann bei Definition leicht vergessen werden
  - Fehlende Semikola führen zu obskuren Fehlermeldungen
- Für Klassen gilt die sogenannte One Definition Rule (ODR):

  No translation unit shall contain more than one definition of any variable, function, class type, enumeration type, or template.
- Definitionen einer Klasse in verschiedenen Kompilierungseinheiten müssen identisch sein

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- Klassen können Daten (Datenmember) und Funktionen (Memberfunktionen) enthalten
- Deklaration von Membern im Rumpf der Klasse wie normale Variablen/Funktionen

Memberfunktionen

- Definition von Memberfunktionen im Klassenrumpf oder außerhalb
- Im Klassenrumpf definierte Funktionen sind implizit **inline**
- Definition außerhalb vom Klassenrumpf:

```
class Interval
{
    ...
    double setEnd(double val);
};
double Interval::setEnd(double val) { this->end   = val; }
```

- **Achtung:** Nicht-inline-Funktionen wegen ODR nicht im Header
- Memberfunktionen **müssen** in der Klassendefinition deklariert werden
- Längere Funktionen üblicherweise in einer `*.cpp`-Datei definieren

Inlinefunktionen

- In C++ können beliebige Funktionen als `inline` deklariert werden:

```
inline bool isGreater(int a, int b) { return a > b; }
```

- Hinweis an den Compiler, bei Verwendung dieser Funktion keinen Funktionsaufruf zu generieren
- Fügt Code der Funktion direkt an der aufrufenden Stelle ein
- Kann Performancevorteile bringen:
  - Spart den Overhead des Funktionsaufrufes
  - Ermöglicht zusätzliche Optimierungen
  - Compiler kann `bool b=isGreater(5, 10);` zu `bool b=false;` optimieren
  - Kann weitere Optimierungen ermöglichen, z.B. bei Verwendung von `b` als Bedingung in einem `if`
- Ersetzt aus C bekannte Verwendung von Makros zur Erzeugung von Funktionen ohne Laufzeitoverhead

Inlinefunktionen II

- `inline`-Funktionen erhöhen die Größe des erzeugten Executables
- Kann Ausführungsgeschwindigkeit durch größere Anzahl Cache-Misses auch verringern
- Aber: Für kleine `inline`-Funktionen kann Executable kleiner werden
  - Für einfache Funktionen Code für den Aufruf möglicherweise größer als der im Rumpf der Funktion ausgeführte Code
  - Durch Inlining ermöglichte Optimierungen können Codegröße reduzieren
- In der Regel sollten nur einfache Funktionen `inline` sein
  - Wenige Codezeilen
  - Schleifen nur mit wenigen Iterationen (besser: Keine Schleifen)

Die Wahrheit über Inlinefunktionen

- Bisherige Aussagen zur Wirkung von `inline` stimmen nicht (mehr)!
- `inline` erzwingt bei heutigen Compilern **kein** Inlining
  - `inline` ist bezogen auf Inlining lediglich ein **Hinweis**, der von modernen Compilern defacto ignoriert wird
  - Compiler darf jede beliebige Funktion inlinen
- Tatsächliche Auswirkung von `inline`:
  erlaubt einen Verstoß gegen die One Definition Rule (ODR)
  - ODR besagt auch, dass jede Funktion im Programm nur einmal (d.h., in nur einer Kompilierungseinheit) definiert sein darf
  - Gilt nicht für `inline`-Funktionen
- Definition von `inline`-Funktionen üblicherweise im Header
- `inline` erlaubt die Existenz mehrerer identischer Definitionen einer Funktion in mehreren Kompilierungseinheiten

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- Memberfunktionen können nur auf Instanzen der Klasse ausgeführt werden
  - Falsch: `Interval.GetStart()` oder `Interval::GetStart()`
  - Korrekt: `foo.GetStart()`, wenn `foo` Variable vom Typ `Interval`

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- Member einer Klasse haben verschiedene Sichtbarkeiten
- Ein Ausdruck *Sichtbarkeit*: steuert den Zugriff auf die folgenden Member

Sichtbarkeiten

- `public`
  - Als `public` deklarierte Member einer Klasse sind öffentlich sichtbar
  - Zugriff von überall möglich
- `private`
  - Auf `private`-Member darf nur aus Methoden der deklarierenden Klasse zugegriffen werden
  - Abgeleitete Klassen erben diese Member, dürfen aber nicht auf sie zugreifen
- `protected`
  - Auf `protected`-Member kann aus Methoden der deklarierenden Klasse, sowie aller von dieser abgeleiteten Klassen zugegriffen werden
- Wird keine Sichtbarkeit angegeben, wird Standardsichtbarkeit angenommen (s. nächste Folie)

Unterschied struct und class

- `struct` und `class` unterscheiden sich in ihrer Standardsichtbarkeit
- `private` für `class`
- `public` für `struct`
- Einziger Unterschied zwischen `struct` und `class` in C++
- Können bei expliziter Angabe der Sichtbarkeit austauschbar verwendet werden
- Verwendung von `struct` und `class` über Konvention festgelegt:
  - `struct` nur für einfache Datentypen ohne Zugriffskontrolle
  - Sonst `class`

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- `this` ist Pointer auf aufgerufene Instanz
- Zugriff auf Member mit `->` Operator
- Mehr zu Pointern in der nächsten Vorlesung

Zugriff auf Member

- In Memberfunktionen Zugriff über `this->`*`Member`*
- Verwendung von *`Member`* ohne `this->` auch möglich
  - Nicht empfohlen, da Zugriff auf Datenmember weniger offensichtlich
  - Klare Angaben über die Herkunft von Variablen und Funktionen erzeugen leichter verständlichen Code
  - Lesbarkeit essentielle Eigenschaft für guten Code
- Zugriff auf Objekt mit dem Dereferenzierungoperator: `*this`
  - `(*this).`*`Member`* äquivalent zu `this->`*`Member`*
  - Klammern notwendig, da `.` stärker bindet als `*`

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- Bereits bekannt: Auf `const`-Objekten dürfen keine Memberfunktionen ausgeführt werden, die das Objekt modifizieren
- Genauer: Nur `const`-Memberfunktionen dürfen ausgeführt werden

Const Member

- `const`-Memberfunktionen dürfen keine Datenmember verändern
- Memberfunktionen können nach `const` überladen werden:

```
class IntContainer {
    std::vector<int> data;
public:
    std::vector<int>&       getUnderlying()
    { return this->data; }
    const std::vector<int>& getUnderlying() const
    { return this->data; }
};
```

- `const`-Version wird nur auf `const`-Objekten ausgeführt
- `.begin()` und `.end()` für Standardcontainer auf diese Weise überladen

Derart überladene Funktionen sollten sich **immer** äquivalent zueinander verhalten!

Eine Intervallklasse

```
class Interval
{
private:
  double start;
  double end;
public:
  Interval(double start, double end) : start(start), end(end)
  {}
  void setStart(double val) { this->start = val; }
  void setEnd(double val)   { this->end   = val; }
  double getStart() const { return this->start; }
  double getEnd()   const { return this->end; }
};
```

- Erstellung von Klasseninstanzen über Konstruktoren
- Konstruktoren sind spezielle Memberfunktionen, die den selben Namen wie die Klasse tragen

Konstruktoren

- Konstruktoren werden **immer** bei Erzeugung eines Objektes aufgerufen
  - Lebenszeit eines Objektes beginnt erst, wenn der Konstruktoraufruf zuende ist.
  - Für primitive Datentypen ist der Standardkonstruktor trivial und führt keinerlei Initialisierung durch
- Konstruktoren werden **nur** zur Erzeugung eines neuen Objektes aufgerufen
  - Aufruf des Konstruktors auf bereits existierenden Objekten ist nicht möglich

Initialisierung von Memberobjekten

- Zweck eines Konstruktors ist die Initialisierung der Datenmember
- Offensichtlicher Ansatz für die Initialisierung:

```
struct Text {
    std::string content;
    Text(const std::string& data) {
        this->content = data;
    }
};
```

- **Aber:** Lebenszeit von Datenmembern beginnt mit Ausführung des Konstruktors
- Im Konstruktorrumpf leben die Datenmember bereits
- Erstellung von Datenmembern vorher durch Aufruf des Defaultkonstruktors

Defaultkonstruktoren

- Ein Defaultkonstruktor ist ein parameterlos aufrufbarer Konstruktor
  - Deklaration von Defaultparametern möglich
- Wird verwendet, wenn kein expliziter Konstruktoraufruf stattfindet
  - `std::string foo;` ruft Defaultkonstruktor von `std::string` auf
  - Initialisiert `foo` mit einem leeren String
- Wird automatisch erzeugt, falls keine Konstruktoren definiert sind
  - Ruft für alle Datenmember den Defaultkonstruktor auf
  - Primitive Datentypen (z.B. int, float) bleiben **uninitialisiert**!
- Keine automatische Erzeugung, ...
  - ... wenn nutzerdefinierte Konstruktoren vorhanden
    Diese Klassen benötigen spezielle Initialisierungen, die der Compiler nicht kennen kann
  - ... wenn Member ohne Defaultkonstruktor vorhanden
  - Defaultkonstruktor muss dann von Hand geschrieben werden, falls einer gewünscht ist

Initialisierung von Memberobjekten

- Zweck eines Konstruktors ist die Initialisierung der Datenmember
- Offensichtlicher Ansatz für die Initialisierung:

```
struct Te
    std::string content;
    Text(const std::string& data) {
        this->content = data;
    }
};
```

Initialisierung von `Content` durch Defaultkonstruktoraufruf

Ersetzen des Inhalts von `Content` durch den von `data`

- **Aber:** Lebenszeit von Datenmembern beginnt mit Ausführung des Konstruktors
- Im Konstruktorrumpf leben die Datenmember bereits
- Erstellung von Datenmembern vorher durch Aufruf des Defaultkonstruktors

Initialisierungslisten

Initialisierung von Memberobjekten besser über Initialisierungsliste (“initialization list”):

```
struct Text
{
    std::string content;
    Text(const std::string& data): content(data) {}
};
```

- Ruft direkt den passenden Konstruktor auf, um `Content` aus einem `std::string` zu initialisieren
  - Sogenannter Copy-Constructor, später mehr dazu
- Effizienter, da Defaultkonstruktion von `Content` gespart wird

Initialisierungslisten II

- Initialisierunglisten sind einzige Möglichkeit für:
  - Memberobjekte, die keinen Defaultkonstruktor besitzen
  - Referenzen als Memberobjekte
    - Zuweisung nicht erlaubt, nur Initialisierung
    - In der Regel keine gute Idee, daher nicht verwenden
    - Alternative: Pointer (aber erst ab der nächsten Vorlesung)
  - Konstante Memberobjekte
- Membervariablen, die nicht explizit in der Initialisierungsliste erwähnt werden, werden defaultinitialisiert
  - Primitive Datenmember bleiben also uninitialisiert!

Aufgrund der höheren Effizienz und der uniformen Anwendbarkeit immer Initialisierungslisten bevorzugen!

Initialisierungsreihenfolge von Memberobjekten

- Initialisierung von Membervariablen in Reihenfolge ihrer Deklaration
  - Nicht in Reihenfolge des Auftretens in der Initialisierungsliste

```
struct Text
{
    unsigned    length;
    std::string content;
    Text(const std::string& data): content(data), length(content.size()) {}
};
```

- Initialisiert `Length` vor der Initialisierung von `Content`
  - Zugriff auf `Content.size()` undefiniertes Verhalten
  - Compiler gibt bei entsprechender Einstellung Warnung aus

Reihenfolge in Initialisierungsliste immer entsprechend der Deklarationsreihenfolge wählen!

initializer_list

- Container wie `std::vector` seit C++11 direkt mit Werten initialisierbar
- Durch Verwendung von `std::initializer_list<T>` auch für eigene Klassen möglich:

```
#include <initializer_list>
struct IntContainer {
    std::vector<int> data;
    IntContainer(std::initializer_list<int> init): data(init) {}
};
IntContainer foo = {1,2,3,4,5};
```

- Genauere Informationen unter:

  http://en.cppreference.com/w/cpp/utility/initializer_list

Uniforme Initialisierungssyntax

- C++11 hat eine neue Initialisierungssyntax eingeführt
  - Als Uniform Initialization Syntax bezeichnet, da Syntax für alle Arten von Initialisierung identisch
  - Abweichungen von bekannter Syntax vor allem in C-Teilen von C++, daher bisher nicht behandelt
- Neue Syntax verwendet geschweifte anstatt von runden Klammern:

```
int foo{5};
Interval bar{2.0, 5.0};
```

- Uniforme Initialisierung hat kein Most-Vexing-Parse
  - Zur Erinnerung: `Interval foo();` definiert Funktion, keine Variable
  - `Interval foo{};` funktioniert problemlos (wenn `Interval` einen Defaultkonstruktor hat)

Uniforme Initialisierungssyntax II

- Häufig ohne explizite Nennung des Typnamens verwendbar
  - Immer, wenn Zieltyp für Compiler klar erkennbar ist

```
Interval foo(Interval x)
{
    x = {1.0, 2.0};
    //...
    return {0.0, 1.0};
}
foo({2.0, 3.0});
```

Uniforme Initialisierungssyntax III

Achtung: Bei Verwendung von uniformer Initialisierungssyntax werden Konstruktoren, die `initializer_list` annehmen, bevorzugt

- `std::vector<int>(10, 1)` konstruiert einen `std::vector` mit zehn Elementen, die alle den Wert 1 haben
- `std::vector<int>{10, 1}` konstruiert einen `std::vector` mit den zwei Elementen 10 und 1

Immer, wenn Argumente des Konstruktoraufrufs implizit in vom Konstruktor akzeptierte `initializer_list` umwandelbar

- `std::vector<double>{10, 1.0}` füllt den Vektor mit 10.0 und 1.0
- `std::vector<std::string>{10, ""}` erstellt einen Vektor, der zehn leere Strings enthält

Implizite Konvertierungen

```
struct Text {
    Text(std::string x);
};
void bar(Text);
...
std::string var{"hello"};
bar(var);
```

- Konstruktoren mit nur einem Parameter definieren implizite Konvertierungen
  - Konstruktor muss mit einem Parameter aufrufbar sein
  - `Text(std::string x, int y = 1)` ermöglicht auch implizite Konvertierungen
- `bar(var)` konstruiert einen `Text` aus `var` und übergibt diesen an `bar`
- Implizite Konstruktion von `std::string` aus Stringliteralen funktioniert auf diese Weise

Explizite Konstruktoren

- Implizite Konvertierung nicht immer erwünscht
  - `std::vector` hat Ein-Parameter-Konstruktor:
    `std::vector<std::string>(5)` erzeugt Vektor mit 5 defaultkonstruierten Strings
  - Implizite Konstruktion aus Integerwerten nicht sinnvoll (und nicht legal):

```
void foo(const std::vector<std::string>& x);
foo(5);
```

- Konstruktoren können als `explicit` deklariert werden
  - Explizite Konstruktoren definieren keine impliziten Konvertierungen

```
struct Text {
    explicit Text(std::string x);
};
void bar(Text);
...
std::string var{"hello"};
bar(var);
```

```
error: could not convert 'var' from 'std::string
{aka std::basic_string<char>}' to 'Text'
```

Nicht-Inline-Konstruktoren

Wie andere Memberfunktionen können Konstruktoren außerhalb des Klassenrumpfes definiert werden:

```
class Interval
{
  ...
  Interval(double start, double end);
};
...
Interval::Interval(double start, double end): start(start), end(end)
{}
```

- Klassenname kommt in der Definition doppelt vor
  - Einmal für den Scope, einmal als Name des Konstruktors
- Wie andere Funktionen auch sollten längere Konstruktoren nicht inline, sondern in der `*.cpp`-Datei definiert werden

Delegating Constructors

- Häufig: Verschiedene Konstruktoren führen fast identische Operationen aus, nur mit anderen Parametern
- In der Vergangenheit durch eine Initialisierungsmethode gelöst, die von den Konstruktoren aufgerufen wird
  - Problematisch, da Konstruktion der Membervariablen notwendigerweise im Konstruktor
- Seit C++11: Delegating Constructors
- Ein Konstruktor kann in der Initialisierungsliste einen anderen Konstruktor aufrufen:

```
Interval::Interval(double start, double end)
  : start(start), end(end)
{}
Interval::Interval() : Interval(0.0, 0.0) {}
```

- Aufruf des anderen Konstruktors muss einziger Eintrag in Initialisierungsliste sein

Friend

- Manchmal ist ein Zugriff auf `private` Daten einer Klasse durch Nichtmemberfunktionen notwendig
- In vielen objektorientierten Programmiersprachen nicht ohne weiteres möglich
  - Führt zu einer erzwungenen Aufweichung der Kapselung
- C++ erlaubt es, Funktionen und Klassen als `friend` einer Klasse zu deklarieren
- `friend`-Funktionen (und Memberfunktionen von `friend`-Klassen) haben dieselben Zugriffsrechte wie Memberfunktionen
  - Dürfen also auf `private` Member einer Klasse zugreifen

Friend II

- `friend` verletzt oberflächlich betrachtet Kapselung
- Daher nur wenn unbedingt nötig anwenden
- Sinnvoll, wenn einige wenige Klassen/Funktionen priviligierten Zugriff auf Member haben sollen
  - Bsp: Factoryklasse, die Instanzen einer Klasse generiert und als einzige Klasse ändernde Memberfunktionen aufrufen darf
- Einzige Alternative: Öffentlicher Zugriff auf entsprechende Member, um einer einzigen Klasse/Funktion den Zugriff darauf zu ermöglichen
- Verwendung von `friend` ermöglicht eine bessere Kapselung

Beispiel Friend

```
class Interval {
    //...
    friend void translateBy(Interval& interval, double val) {
        interval.start += val;
        interval.end   += val;
    }
    friend class Foo;
};

Interval area(1.0, 5.0);
translateBy(area, 5.0);

class Foo {
    //...
    double bar(Interval i)
    { return i.start + i.end; }
};
```

Achtung: Definiert kein Member, sondern eine freie Funktion, auch wenn es im Klassenrumpf steht

Namespaces

```
namespace Simulator {
  class Cell {...};
  ...
}
```

- Anlegen eigener Namespaces mit `namespace` *Name* `{...}`
- Namespaces können beliebig geschachtelt werden:
  `namespace Simulator { namespace Parser {...}}`
- Definitionen von Memberfunktionen nur im selben Namespace wie die Klasse
- Forward-Deklarationen müssen jeweils im Namespace des deklarierten Objekts sein:

```
namespace Simulator {
  class Cell;
}
//code using the forward-declaration of Cell
```

Namespace-Aliases

- Mit `namespace` *Name* = *SomeName*; können Aliases für Namespaces gesetzt werden
  - `namespace Sim = Simulator;` definiert beispielsweise `Sim` als Alias für `Simulator`
- Namespace Aliases können wie der Namespace verwendet werden
  - Im Beispiel kann auf `Simulator::Cell` also auch über `Sim::Cell` zugegriffen werden
- Hilfreich zur Abkürzung von verschachtelten Namespace
  - Bsp: `namespace SParser = Simulator::Parser;`

Using Namespaces

- `using namespace` *Name*; importiert alle Symbole (Klassen, Funktionen, Variablen) aus *Name* in den aktuellen Namespace:

```
using namespace Simulator;
foo(const Cell&);//Shorthand for Simulator::Cell
```

- Häufig in Tutorials und Beispielcode verwendet
  - Insbesondere `using namespace std;`
- In echtem Code eher vermeiden
  - Name-Lookup ist in C++ relativ komplex (später mehr)
  - Ohne genaue Kenntnisse der Regeln kann es schwierig sein, die Herkunft eines Symbols zu bestimmen
  - Klarheit über die Herkunft von Symbolen hilft der Lesbarkeit
  - `using namespace` macht Name-Lookup noch komplexer

`using namespace` nicht verwenden!

Enums

- Typen zum Abbilden einer begrenzten Zahl von Möglichkeiten (Aufzählungstypen)
- Zwei Arten von Enums: `enum` und `enum class`
- `enum` ältere, von C übernommene Variante
- `enum class` (oder äquivalent: `enum struct`) seit C++11
- `enum class` bietet einige Vorteile gegenüber `enum`
  - Typsicherheit
  - Eigener Scope

In der Regel `enum class` bevorzugen

Enum

```
enum class Corner {
    TopLeft,
    TopRight,
    BottomLeft,
    BottomRight
};
```

- Enums sind Integertypen, die nur bestimmte Werte annehmen können
- Mögliche Werte werden in Definition in einer Komma-separierten Liste festgelegt
- Hier: Variable vom Typ `Corner` kann die Werte `TopLeft`, `TopRight`, `BottomLeft` und `BottomRight` annehmen
- Zugrundeliegende Integerwerte werden automatisch festgelegt

Enum

```
enum class Corner {
    TopLeft     = 1,
    TopRight    = 2,
    BottomLeft  = 4,
    BottomRight = 8
};
```

- Wert kann auch mit "=*value*" festgelegt werden
- **Achtung:** Zugrundeliegender Wert Implementationsdetail
  - In verwendendem Code nicht darauf verlassen
- Auch möglich, nur einen Teil der Werte festzulegen

Basistyp von Enums

- Enums haben einen zugrundeliegenden Integertypen (Basistyp)
- Festlegung des Basistyps über “: *type*”:

```
enum class Corner: short
{ TopLeft, TopRight, BottomLeft, BottomRight };
```

- Bei fehlender Angabe wird `int` als Basistyp verwendet
- Basistyp muss alle Enumwerte abbilden können
  - `enum class Foo: char { Bar = 1024};` ist kein gültiger Enum

Verwendung von Enums

```
enum class Corner
{
    TopLeft,
    TopRight,
    BottomLeft,
    BottomRight
};
Corner corner = Corner::TopRight;
```

- Enums bilden einen eigenen Scope
- Auf Enumnamen wird mit *Type*::*Value* zugegriffen
- `enum class` und `enum struct` sind typsicher
  - Keine implizite Konvertierung zu/von dem Basistyp

Allgemeines zu den Aufgaben

Allgemeine Hinweise zu den Aufgaben

- Compilergenerierte Dateien gehören nicht in das Repository
  - Gilt vor allem für `*.o`, `*.d` Dateien, sowie die generierte Executable
  - `.gitignore`-Datei im Repository sollte dies eigentlich verhindern...
  - solange die Dateienamen den Vorgaben entsprechen
- Variablennamen
  - Die Basissprache für Programmierung ist Englisch
  - Bezeichner (Variablennamen, ...) sollten daher auch Englisch sein
  - Bei Mitarbeit an größeren Projekten wahrscheinlich vorgeschrieben
  - Am besten von vorneherein angewöhnen
  - Selbes gilt für Kommentare

Allgemeine C++-Anmerkungen

- Für Vectoren `operator[]` gegenüber `.at()` bevorzugen
- `.emplace_back(`*`args`*`)` ist effizienter und kürzer als `.push_back(make_tuple(`*`args`*`))`
- Für Iteratoren `auto` zur Abkürzung der Typnamen verwenden
  - `auto iter = vec.begin();`
  - `std::vector<std::string>::iterator iter = vec.begin();`
- ++*var* gegenüber *var*++ bevorzugen
  - *var*++ gibt den Wert vor der Inkrementierung zurück, müsste also das inkrementierte Objekt kopieren
  - Wird Rückgabewert nicht benutzt, dann möglicherweise Performanceoverhead
  - Aktuelle Compiler optimieren das zwar für einfache Zahlen, es ist aber besserer Stil

Programmierstil in C++

- Für die Softwareentwicklung ist Wartbarkeit ein wichtiges Qualitätsmerkmal
  - Code muss möglichst einfach zu lesen sein
- Für C++-Programme ist Effizenz häufig ebenfalls wichtig
  - Kein Kernthema dieser Veranstaltung
  - Einige Performancesünden sollten dennoch vermieden werden
- Entwicklung effizienter und wartbarer Software benötigt vor allem Übung
- Einige Regeln, die dies in C++ erleichtern, sind auf den nächsten Folien aufgeführt
- **Grundlose** Nichtbeachtung dieser Regeln kann in den folgenden Übungsblättern zu Punktabzug führen!

Regeln für die C++-Programmierung

- Range-based-For ist eleganter als ein Iterator-Loop
  - `for(const std::string& str: vec)`
  - `for(auto iter = vec.begin(); iter != vec.end(); ++iter)`
  - Kürzer und für den Leser leichter nachzuvollziehen
  - Immer verwenden, wenn über einen kompletten Container iteriert werden soll!
- Variable im Range-based-For immer als Referenz deklarieren!
  - Erzeugt ansonsten unnötige Kopien
  - `for(const T& val: vec)` statt `for(T val: vec)`
  - Wenn möglich: Bevorzugt `const` verwenden
- Iteratorinkrementierung nur zum Iterieren und für notwendige Anpassungen der Laufvariable verwenden!
  - Verwendung von Iteratoren zum Adressieren fester Elemente kann den Code deutlich unübersichtlicher gestalten
- Keine C-Style-Arrays verwenden!
  - Bessere Alternative: Standardcontainer

Regeln für die C++-Programmierung II

- Aussagekräftige (englische) Variablennamen verwenden!
  - Wichtig für das Verständnis des Codes
  - `temp` gehört nicht dazu!
  - Variablennamen mit einem Buchstaben nur für Laufvariablen von Schleifen und in Spezialfällen, z.B. `x` und `y` für Koordinaten, verwenden
- Variablen erst zum Zeitpunkt der Initialisierung definieren!
  - Übersichtlicher, da Variablen in möglichst kleinem Scope existiert
  - In der Regel effizienter, da das Objekt sonst erst defaultkonstruiert wird, um es später zu überschreiben
  - Ausnahme: Die Variable wird außerhalb des initialisierenden Scopes benötigt
  - Ausnahmen bzgl. der Effizienz können für Schleifen gelten, später mehr dazu

Nichtbeachtung ohne **überzeugende** Begründung kann zu Punktabzug führen!