Programmieren in C++

C++20 Kompakt

Modernes C++

- C++11
  - move semantic
  - smart pointers
  - threading & asynchronism
  - lambda expressions
  - initializer lists
  - range based for loops
  - variadic templates
  - type deduction
  - compile time assertions
- C++14
  - generic lambda expressions
  - user defined literals
  - return type deduction
  - make_unique
- C++17
  - parallel algorithms
  - fold expressions
  - class template deduction
  - file system library
  - variant, optional, …
  - string_view
- C++20
  - concepts
  - Ranges-Bibliothek
  - Module
  - Coroutinen

Entwicklung von Klassen

- Klassische Aufteilung in
  - öffentliche Schnittstelle: h-Datei
  - Implementierung der Methoden: cpp-Datei
- Schnittstellendatei
  - definiert die Klasse (Attribute, Konstruktoren, Methoden, Operatoren, …)
  - kann inline programmierte Prozeduren enthalten
  - kann andere benötigte Schnittstellen inkludieren (#include)
- Implementierungsdatei (mehrere pro Klasse möglich)
  - inkludiert zugehörige Schnittstelle
  - kann weitere benötigte Schnittstellen inkludieren
  - implementiert die in der Schnittstelle beschriebenen Prozeduren
- header-only Software-Bibliotheken (Open Source)
  - der ganze Code wird in hpp- bzw. h-Dateien entwickelt und als Quellcode ausgeliefert

Klassendeklarationen

- Öffentliche Klasse

```
struct Point {
  int m_x, m_y;  // öffentliche Attribute (Members, Instanzvariablen)
  double dist(Point p) const;// in C kann ein struct nur Attribute enthalten
};
```

- Klasse

```
class Person {
  std::string m_name;     // private Attribute (Members, Instanzvariablen)
  int m_age;
public:           // ab hier wird die Sichtbarkeit auf public geändert
  Person(const char name[], int age); // Konstruktor
  std::string getName() const;
  void setAge(int age);
};
```

Klassenimplementierung und -nutzung

- Klasse Point

```
int Point::dist(Point p) const {
    int dx = p.m_x – m_x;
    int dy = p.m_y – m_y;
    return hypot(dx, dy);
}
```

- Lokale Instanzen erstellen

```
Point pnt1;     // liegt auf dem Stack
Point pnt2;     // liegt auf dem Stack
```

- Zugriff auf Instanzvariable

```
pnt.m_x = 3;
```

- Aufruf einer Instanzmethode

```
double d = pnt1.dist(pnt2);
```

- Klasse Person

```
Person::Person(const char name[], int age)
    : m_name(name), m_age(age)
{}
Person::getName() const {
    return m_name;
}
```

- Lokale Instanz erstellen

```
Person pers("Peter", 21); // lauf dem Stack
```

- Aufruf von Instanzmethoden

```
pnt.setY(7);
// Stringobjekt wird kopiert (tiefe Kopie)
string s = pers.getName();
```

Automatische Typinferenz

- Schlüsselwort auto
  - bei Variablendefinitionen, wo aus dem Initialisierungswert der Variable der Typ der Variable für den Compiler automatisch ersichtlich ist, kann das Schlüsselwort auto anstatt des konkreten Typs hingeschrieben werden
  - Beispiele

```
auto x = 7;
double f();
auto g = f();
```

- Schlüsselwort decltype
  - decltype(x) ist eine Funktion, welche den Deklarationstyp des Ausdruckes x zurückgibt
  - Beispiele

```
decltype(8) y = 8;
decltype(g) h = 5.5;
```

Schlüsselwort constexpr

- Konstanter Ausdruck
  - ein Ausdruck, dessen Wert bereits zur Kompilationszeit bestimmt wird
  - darf nur aus Literalen und anderen constexpr Werten bestehen
  - Beispiele

```
constexpr size_t Length = 500;
constexpr size_t L2 = Length*Length/4;
constexpr char Grades[] = {'A', 'B', 'C', 'D', 'E', 'F' };
double constexpr Pi = 3.141596;
```

- Konstante Funktionen
  - eine Funktion, welche prinzipiell zur Kompilationszeit ausgeführt werden kann und einen constexpr Wert zurückliefert
  - Iteration und Rekursion sind erlaubt
  - die Funktion kann aber auch zur Laufzeit ausgeführt werden
  - Beispiele

```
constexpr int sum(int x) {
   int sum = 0;
   for (int i = 0; i <= x; i++) sum += i;
   return sum;
}
```

Nichtveränderbare Speicherzellen

- Schlüsselwort const
  - Unveränderbarkeit: nach Initialisierung nur noch lesender Zugriff
- Beispiele

```
const auto age = pers1.getAge();
vector<int> v = { 1, 2, 3, 4, 5, 6 };
const size_t size = v.size();
size_t strlen(const char s[]) { … }
void print(const string& s) { … }
```

- const darf auch nach dem Typ stehen

```
double const PI = computePi();
auto const PID2 =  PI/2;
```

Vereinheitlichte Initialisierung

```
struct Base { };
struct Derived : public Base {
    int m_member;
    Derived(int a1, int a2) : Derived{a1 + a2} {}
    Derived(int a) : Base{}, m_member{a} {}
};
struct Triple {
    int a, b, c;            // Members sind öffentlich, kein Konstruktor vorhanden
};
Derived obj1{1, 2};         // Alternative: obj1(1, 2)
Derived obj2 = {1, 2};      // = funktioniert nur weil der Konstruktor nicht explizit ist
auto *p = new Derived{1, 2};// Alternative: new Derived(1, 2)
vector<int> vec = {1,2,3,4};// vector bietet einen ctor mit Initialisierungsliste an
Triple t = {7, 8};          // kein Konstruktoraufruf, sondern Aggregat-Initialisierung
                            // bei zu wenig Werten werden die restlichen Members
                            //  (hier c) mit 0 initialisiert
```

Initialisierungslisten

- Initialisierungslisten sind ein generischer Typ

```
#include <initializer_list>
struct Tuple {
   int value[];
   Tuple(const initializer_list<int>& v);              // ctor #1
   Tuple(int a, int b, int c);                         // ctor #2
   Tuple(const initializer_list<int>& v, size_t cap); // ctor #3
};
Tuple t1(4, 5, 6);                  // ctor #2 wird verwendet
Tuple t2{1, 2, 3};                  // ctor #1 wird verwendet
Tuple t3{2, 4, 6, 8};               // ctor #1 wird verwendet
Tuple t4{{2, 4, 6}, 3};             // ctor #3 wird verwendet
```

- Randbedingungen
  - wenn die Initialisierungsliste der einzige Parameter ist, kann wie oben gezeigt vorgegangen werden
  - wenn noch weitere Parameter vorhanden sind, dann müssen die geschweiften Klammern verschachtelt werden

Typkonvertierung im Überblick

- C
  - Syntax: (type)expression
- C++
  - static_cast: normale Typkonvertierung

```
int x = static_cast<int>(2.0);
```

  - dynamic_cast: down-cast in Klassenhierarchie

```
Base *b = new Derived(21);
Derived *d = dynamic_cast<Derived*>(b);
```

  - const_cast: const hinzufügen oder entfernen

```
const Point p;
const_cast<Point&>(p).setX(4);
```

  - reinterpret_cast: keine Compiler-Checks

```
float f = 3.14f;
int bitRepresentation = *reinterpret_cast<int*>(&f);
```

Klassendeklarationen

- Öffentliche Klasse

```
struct Point {
  int m_x, m_y;  // öffentliche Attribute (Members, Instanzvariablen)
  double dist(Point p) const;      // in C kann ein struct nur Attribute
};                                 // enthalten
```

- Klasse

```
class Person {
  std::string m_name;     // private Attribute (Members, Instanzvariablen)
  int m_age;
public:             // ab hier wird die Sichtbarkeit auf public geändert
  Person(const char name[], int age); // Konstruktor
  std::string getName() const;
  void setAge(int age);
};
```

Objekterzeugung

```
Person p; // globales Punktobjekt (automatisch mit 0 initialisiert)

int main(int argc, char *argv[]) {
   Person *pPers = nullptr;   // Zeiger auf Person (mit null init.)

   Person tom("Tom", 21);     // Personenobjekt auf dem Stack
   Person tom2 = tom;         // tiefe Kopie von tom
   Person tom3(tom);          // tiefe Kopie von tom

   pPers = new Person("Anna", 20);       // pPers zeigt auf
                                         // Personenobjekt auf dem Heap
   delete pPers;   // gibt den Speicher für das Personenobjekt auf
                   // dem Heap wieder frei
}
```

Lebensdauer von Objekten

- Statischer Speicher
  - Globale Variablen und Modulvariablen bleiben während der ganzen Laufzeit des Programms im Speicher
- Dynamischer Speicher (Heap)
  - nicht mehr benötigte Objekte sollten mit delete freigegeben werden zur Vermeidung von Memory-Leaks
  - nicht mehr benötigte C-Arrays/C-Strings sollten mit delete[] freigegeben werden
  - bei Terminierung des ausführenden Prozesses wird aller Speicher freigegeben
- Automatischer Speicher (Stack)
  - auf dem Stack angelegte lokale Variablen und Parameter werden beim Verlassen des Blocks automatisch vom Stack entfernt
  - Variablen so lokal wie möglich definieren, damit sie möglich spät erstellt oder möglichst früh wieder freigegeben werden

Zeiger und Adressoperator

- Zeiger (Pointer)
  - ein Zeiger zeigt auf eine Speicherstelle des (virtuellen) Adressraums
  - Speicherbedarf eines Zeigers: x86: 32 Bit, x64: 64 Bit
  - Zeiger sind stark typisiert
  - von jeder Variable, Funktion, Methode und jedem Objekt kann mit dem Adressoperator & zur Laufzeit die Adresse (Speicherstelle) abgefragt werden; das Resultat einer solchen Abfrage ist ein Zeiger
  - über den Dereferenzierungsoperator * kann vom Zeiger auf die Variable, Funktion, Methode oder das Objekt zugegriffen werden
  - Java: eine Referenz in Java entspricht etwa einem Zeiger in C++
- Einfaches Beispiel

```
int x = 5;          // x wird direkt mit 5 initialisiert
int* px = &x;       // initialisierter Zeiger auf Integer x
*px = 7;            // Wert von x wird indirekt auf 7 gesetzt
```

Eigenschaften von Zeigern

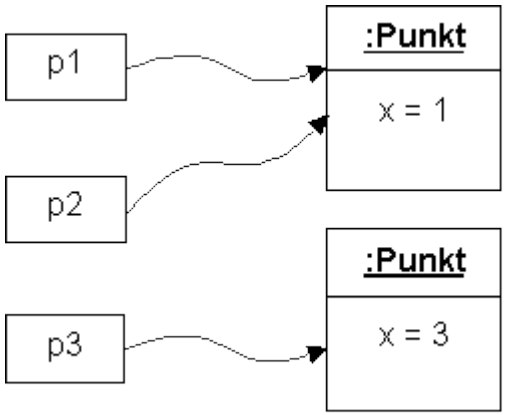
- haben einen Typ „Zeiger auf …“
  - soll eine Zeigervariable auf eine Instanz einer Klasse C zeigen, so muss der Typ der Zeigervariablen zur Klasse C zuweisungskompatibel sein

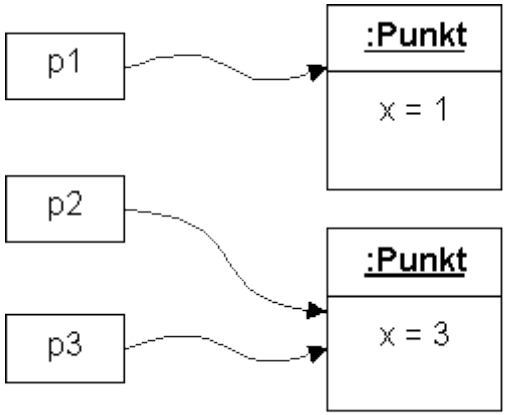
```
Point pnt;
Person pers("Peter", 21);
Point *p1 = &pnt;
Point *p2 = &pers;          // nicht zuweisungskompatibel
```

- zeigen auf gültige Speicheradressen, z.B.
  - dynamisch allozierte Objekte auf dem Heap
  - aufs erste Element von C-Arrays bzw. C-Strings
  - auf statische Variablen und Objekte (Achtung Lebensdauer!)
- zeigen auf ungültige Speicheradressen
  - nullptr
  - nicht initialisierten Speicherbereich

Zuweisungen bei Zeigervariablen

```
Point *p1 = new Point();  // p1 zeigt auf neu erstelltes Objekt auf dem Heap
Point *p2;                // p2 ist ein nichtinitialisierter Zeiger
Point *p3 = new Point();  // p3 zeigt auf neues Point-Objekt auf dem Heap
p2 = p1;                  // p2 zeigt zum gleichen Objekt wie p1
p2 = p3;                  // Adresse p3 wird nach p2 kopiert
// Achtung: die Punktdaten werden hier nicht kopiert
```





Instanzen und Instanzmethoden

- Erzeugen von Klasseninstanzen

  Beispiele

```
Point pnt1;                          // Punktobjekt auf dem Stack
Person pers1("Anna", 22);            // Person auf dem Stack
Point *pnt2 = new Point();           // Punktobjekt auf dem Heap
Person *pers2 = new Person("Urs", 21);   // Person auf dem Heap
```

- Zugriff auf Instanzvariablen bzw. Instanzmethoden

  Beispiele

```
pnt1.m_x = 3;    // direkter Zugriff auf Instanzvariable m_x
(*pnt2).m_x = 4; // indirekter Zugriff auf Instanzvariable m_x
pnt2->m_x = 5;   // vereinf. Schreibweise des indirekten Zugriffs
auto d1 = pnt1.dist(*pnt2);
auto d2 = pnt2->dist(pnt1);
```

Referenzen (C++)

- Referenzen
  - sind Aliasse für andere Variablen (sog. lvalue)
  - haben keine eigene Repräsentanz im Speicher
  - werden durch ein & gekennzeichnet
  - müssen immer initialisiert werden (Neuinitialisierung ist unmöglich)
  - vereinfachen die effiziente Parameterübergabe (keine Datenkopie)

- Beispiele

```
int k = 2;
int& rk = k;        // rk ist ein Alias für k
rk = 3;             // die Variable k kriegt den Wert 3

Point pnt1;
auto& rx = pnt1.m_x;      // rx is ein Alias für das
                          // Attribut m_x von pnt1
rx = 5;             // Attribut m_x von pnt1 erhält den Wert 5
```

Zeiger und Referenzen

```
int x;
int& rx = x;     // rx ist ein Alias für die Variable x
int* px = &x;    // px ist ein Zeiger auf x
px = &rx;        // px ist auch ein Zeiger auf x

int k;
int *pk = &k;    // das Ampersand (&) ist der Adressoperator
int*& rpk = pk; // rpk ist ein Alias für den Zeiger pk
*rpk = 4;        // die Variable k kriegt den Wert 4

int a = 2, b = 9;
int *pa = &a, *& rpa = pa;
*rpa = 4; rpa = &b; pa = &a;
cout << *rpa << endl;       // welcher Wert wird ausgegeben?
```

Parameterübergabe

- In welcher Art können Objekte an Methoden übergeben werden?
  - By Value

```
void foo1(int x)            // by value: Daten (auch Zeiger) werden kopiert
```

  - By Reference

```
void foo3(const Person& p)// in: referenzierte Person wird nicht kopiert
void foo4(Person& p)      // in-out: referenzierte Person wird nicht kopiert
                          // kann aber in foo4 verändert werden
```

  - By Pointer

```
void foo5(Point* p)       // good-practice: nur für out-Parameter
                          // verwenden, da beim Aufruf der out-Parameter
                          // gut über den Adressoperator erkennbar ist
```

- Good Practice
  - Datentypen mit weniger oder gleichviel Speicher wie zwei Zeiger werden üblicherweise by value übergeben

Rückgabetypen

- By Value
  - Daten werden in Form eines temporären Objekts zurückkopiert

    ```
    double sqrt(double x)
    Point move(const Point& p, int dx) // Ansatz: Point is immutable
    ```
  - bei grossen Objekten effizientere in-out oder out Parameterübergabe nutzen
- By Reference und By Pointer
  - darf nur verwendet werden, wenn die Referenz bzw. der Zeiger auf das zurückgegebene Objekt eine längere Lebensdauer als die Übergabeparameter hat

    ```
    Point& Point::move(int dx, int dy) { … return *this; }// Point is mutable
    ```
  - entspricht der Rückgabe eines impliziten Zeigers
  - falsche Verwendung (verwendet impliziten Zeiger auf zerstörtes Objekt)

    ```
    Point& createPoint(int x, int y) { Point p(x, y); return p; }
    ```

Smart Pointers (C++)

- Prinzip
  - spezielle Zeigerobjekte verwalten Adressen
  - mittels Referenzzähler wird festgehalten, wie viele Zeigerobjekte auf das gleiche Objekt auf dem Heap zeigen
  - im Destruktor des Zeigerobjektes wird der Referenzzähler überprüft und das Objekt auf dem Heap automatisch gelöscht, wenn keine weiteren Zeigerobjekte mehr auf das gleiche Objekt zeigen
- Ziel
  - der Umgang mit den Zeigerobjekten soll so einfach sein, wie der Umgang mit Rohzeigern, d.h. der Benutzer soll nichts mit dem Referenzzähler zu tun haben
  - Verzicht auf explizite Speicherallokation (`new`) und -freigabe (`delete`)
- Vorteil gegenüber Garbage Collector (Performanz)
  - Speicher wird sofort frei gegeben, sobald er nicht mehr benötigt wird
  - keine aufwendige Suche von nicht mehr benötigten Objekten
  - Umgang funktioniert so einfach wie bei lokalen Objekten auf dem Stack

Ownership-Konzept

- Heap-Objekt hat genau einen Besitzer
  - `std::unique_ptr<T>`
    - pro Objekt existiert höchstens ein einziger Besitzer
    - unique_ptr ist der Besitzer des Objektes, auf welches verwiesen wird
    - wird als Returntyp von Factories verwendet
    - das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts zerstört
- Heap-Objekt kann mehrere Besitzer haben
  - `std::shared_ptr<T>`
    - mehrere Zeigerobjekte können auf das gleiche Objekt zeigen
    - shared_ptr benutzt Referenzzähler
    - das Objekt wird beim Aufruf des Destruktors des Zeigerobjekts nur dann zerstört, wenn keine weiteren shared_ptr aufs gleiche Objekt zeigen
  - `std::weak_ptr<T>`
    - wie shared_ptr, aber ohne Referenzzähler
    - wird zum manuellen Aufbrechen von zyklischen Abhängigkeiten benötigt

Eindimensionale C-Arrays

- Grundsätze
  - Länge des Arrays wird nicht im Array abgespeichert
  - die Länge ist dem Compiler nur im Sichtbarkeitsbereich der Definition des Arrays bekannt
  - sehr grosse Arrays sollen auf dem Heap (dynamisch) angelegt werden
- statische Erzeugung
  - Wenn die Arraylänge zur Kompilationszeit bekannt und konstant ist, dann kann das Array auf dem Stack angelegt werden
  - Beispiel

```
char text[100];
```

- dynamische Erzeugung
  - Array wird zur Laufzeit auf dem Heap angelegt
  - Beispiel

```
int len = …;                          // len kann, muss aber nicht konstant sein
char * const text = new char[len]; // new liefert einen konstanten Zeiger zurück
delete[] text;                        // Speicherplatz des Arrays wird freigegeben
```

C-Strings

- Was ist ein C-String?
  - ein eindimensionales Character-Array mit 0-Terminierung
  - Ende der gültigen Zeichenkette ist durch ein '\0'-Character gekennzeichnet
  - die 0-Terminierung benötigt ein zusätzliches Byte
- Unterschied zu anderen Arrays
  - vereinfachte Initialisierung erlaubt
    - char s[] = "Das ist ein Test."; // String-Schreibweise anstatt Initialisierungsliste
    - s zeigt auf eine Kopie des String-Literals "Das ist ein Test."
    - sizeof(s) gibt den Speicherbedarf des Strings nur im Sichtbereich der Definition zurück, ausserhalb wird der Speicherbedarf des Zeigers s zurückgegeben
  - implizite Konstante
    - const char *t = "Das ist ein Test.";
    - t zeigt direkt auf den konstanten String der Länge 17 + 1 Byte für 0-Terminierung
    - sizeof(t) gibt die Anzahl Bytes des Zeigers t zurück

Zeigerarithmetik

- Voraussetzung
  - Zeigertyp: in einer Zeigervariablen (Zeiger) wird eine Speicheradresse verwaltet
- Idee
  - aus bestehender Speicheradresse wird eine neue Adresse berechnet
- Erlaubte Operationen
  - +, +=, ++
  - -, -=, --
  - Ergebnis ist vom gleichen Zeigertyp
  - +1 bedeutet nicht + 1 Byte, sondern + Anzahl Bytes des Zielobjekts des Zeigers
- Typischer Einsatz
  - durch die Bildpunkte eines Rasterbildes (Arrays) iterieren

C++ Arrays (statisch)

- class array<T,S> mit fester Grösse S
  - generische Klasse aus der STL
  - kapselt ein C-Array fixer Länge und bietet ein paar nützliche Array-Methoden
  - keine Unterscheidung zwischen Array-Länge und Kapazität
- Beispiel

```
#include <array>
#include <string>

constexpr size_t size = 4;
array<string, size> names = { "adam", "berta", "carlo", "doris" };
array names2 { "adam", "berta", "carlo", "doris" };
int i = 0;
for (const auto& s : names) {
   cout << i++ << ": " << s << endl;
}
```

C++ Vektoren (halbdynamisch)

- class vector<T>
  - generische Klasse aus der STL, entspricht der ArrayList aus Java
  - Unterscheidung zwischen Länge und Kapazität

- Beispiel

```
vector<string> vnames = { "adam", "berta", "carlo", "doris" };
vector<shared_ptr<string>> vsp;
vsp.reserve(vnames.size());// allocates enough memory on heap

for (const auto& s : vnames) {
   vsp.push_back(make_shared<string>(s + ':' +
      to_string(s.length())));
}
for (size_t i = 0; i < vnames.size(); i++) {
   cout << vnames[i] << endl;
   cout << *vsp[i] << endl;
}
```

C++ Strings

- class string ist gleich basic_string<char>
  - #include <string>
- Beispiel

```
auto name = "Andrea"s;          // ist ein C++-String und kein C-String
                                // wegen dem suffix s

name.size();                    // Anzahl Zeichen
name.length();                  // Anzahl Zeichen
name[2];                        // direkter Zeichenzugriff
name.c_str();                   // Konverter zu nullterminiertem C-String
name.begin();                   // Iteratoren
name.substr(2, 2);              // Substring(pos, len)
name.find("re");                // Suchalgorithmus
```

C++ string_view

- Wrapper für ein String-Literal und seine Länge
  - besteht üblicherweise nur aus zwei Attributen
    - const char* data; // Zeiger zu einer konstanten Zeichenkette
    - size_t size; // Anzahl Zeichen
- Eigenschaften
  - das string_view Objekt ist nicht der Besitzer der Zeichenkette
    - die Zeichenkette wird nicht von string_view angelegt
    - der Speicher der Zeichenkette wird nicht freigegeben
  - die Konstruktoren und Methoden benötigen keine Ausführungszeit
    - die Objekte und Rückgabewert sind constexpr
- Einsatz
  - kann als effizienter Ersatz eines C-Strings verwendet werden
  - die Länge des C-Strings muss nicht separat an eine Methode übergeben werden

C++ span<T>

- Kapselung einer Sequenz von Daten und deren Länge
  - typische Datensequenzen sind:
    - C-Array
    - std::array
    - std::vector
  - ähnliches Prinzip wie bei string_view
    - Daten der Sequenz dürfen aber verändert werden
  - typischer Einsatz
    - zur Datenübergabe an Funktionen
- Beispiele
  - span<int> s1; // Sequenz von int's
  - span<const int> s2; // Sequenz von nicht veränderbaren int's
  - span<int, 5> s3; // Sequenz von exakt 5 int's

C++ Zeichentypen

- Standardzeichentypen (in Standardbibliothek voll unterstützt)
  - const char *s = "abcd"; 1 Byte pro char
  - const wchar_t *s = L"αβγδ"; mehrere Bytes pro Character (z.B. UTF-16)
- String-Repräsentationen
  - const char8_t *s = u8"αβγδ"; UTF-8 String-Repräsentation
  - const char16_t *s = u"αβγδ "; UTF-16 String-Repräsentation
  - const char32_t *s = U"αβγδ "; UTF-32 String-Repräsentation
- Unicode-Codepoints
  - 16 Bit Unicode-Codepoints: \u1234 (4-stelliger Hex-Code)
  - 32 Bit Unicode-Codepoints: \U00123456 (8-stelliger Hex-Code)

C++ String-Typen

```
// UTF-8 (neuer MSVC-Standard)
string s = "ab\u1234\U00103456äö👦👧";

// mehrere Bytes pro Character
wstring s = L"ab\u1234\U00103456äö👦👧";

// UTF-8 String-Repräsentation
u8string s = u8"ab\u1234\U00103456äö👦👧";

// UTF-16 String-Repräsentation
u16string s = u"ab\u1234\U00103456äö👦👧";

// UTF-32 String-Repräsentation
u32string s = U"ab\u1234\U00103456äö👦👧";
```

Aufzählungsklassen

- Syntax einer stark typsicheren Aufzählungsklasse

  enum class *Typname* [: *BasisTyp*] { *Liste möglicher Werte* } [*Variablenliste*] ;

- Beispiele

```
// Standardwerte 0, 1, 2, 3, .., 7 werden verwendet
enum class Color : uint8_t {
    black, red, green, yellow, blue, magenta, cyan, white };
Color c1 = Color::white;
Color c2 = (Color)1;
if (Color::red != Color::white) …

enum class Vehicle { bicycle, car, bus, train };
using enum Vehicle; // C++20
Vehicle v = car;
using BaseType = std::underlying_type<Vehicle>::type;
// int typeid(BaseType).name()
BaseType b = 10;
~~Vehicle v2 = 10;~~
```

Konstruktoren

- primitive Datentypen besitzen keine Konstruktoren
- Konstruktoren heissen gleich wie die Klasse und initialisieren die Attribute eines Objekts
  - aggregierte Objekte werden durch zugehörige Konstruktoren initialisiert
  - primitive Attribute müssen initialisiert werden (keine automatische Initialisierung)
- können nur bei der Erzeugung von Objekten mit gleichzeitiger Initialisierung aufgerufen werden (kann nicht zur Re-Initialisierung verwendet werden)

Beispiel

```
class Point {
    // implementierter Standard-Konstruktor
    Point() : m_x(0), m_y(0), m_z(0), m_color(Color::black) { }

    // implementierter benutzerdefinierter Konstruktor (Farbe: standardmässig blau)
    Point(double x, double y, double z)
    : m_x(x),  m_y(y), m_z(z), m_color(Color::blue)
    { }
}
```

Vorgabeparameter (Default-Parameter)

- Parameter in Methoden dürfen mit Standardwerten belegt werden
  - Default-Parameter werden nur in der Schnittstelle angegeben
- für Default-Parameter müssen beim Methodenaufruf keine Werte angegeben werden (es dürfen aber)
- in der Parameterliste einer Methode müssen
  - zuerst alle Parameter ohne Default-Wert
  - dann alle Parameter mit Default-Wert
- aufgelistet werden
- alle Methoden und Konstruktoren dürfen Default-Parameter verwenden
- Beispiel: verbesserter Konstruktor mit voreingestellter Farbe

```
Point(double x, double y, double z, Color color = Color::blue)
    : m_x(x), m_y(y), m_z(z), m_color(color)
{ }
```

Klassen: Schlüsselwort const

```
class Ray {
    const Point m_origin;
    Point m_onRay;
public:
    Ray(const Point& p) : m_origin(p), m_onRay(p) { }
    void setPointOn(double x, double y, double z);
    Point getPoint() const { return m_onRay; }
};
```

nicht veränderbar
muss mit Standardwert versehen oder in der Initialisierungsliste initialisiert werden

p ist unveränderbar

m_onRay ist unveränderbar in dieser Methode

Einsatz

```
const Ray ray(p3);          // ray darf nicht modifiziert werden
ray.setPointOn(1, 3, 5);    // daher ist schreibender Zugriff nicht erlaubt
Point p = ray.getPoint();   // ok, da getPoint() nur lesend zugreift
```

this-Zeiger

- zeigt auf die eigene Instanz
- wird in Instanzmethoden verwendet
- Suizid: delete this;

Beispiel

```
Point& move(double d[3]) {
    m_x += d[0];
    m_y += d[1];
    m_z += d[2];
    return *this;
}
```

Anwendung

```
double delta[] = { 1, 2, 3 };
Point p(0, 0, 0);
p.move(delta).move(delta);          // ergibt Koordinaten (2, 4, 6)
```

Klassenvariablen und -methoden

- Klassenvariablen
  - werden pro Klasse und nicht pro Instanz angelegt
  - alle Instanzen einer Klasse haben Zugriff auf die gemeinsamen Klassenvariablen dieser Klasse
  - Modifikator static vor dem Typ der Variable
  - Einsatzmöglichkeiten
    - zählen der erzeugten Instanzen einer Klasse
    - Registrierung des zuletzt erzeugten Objektes
    - Konstanten
- Klassenmethoden
  - können ohne Instanz einer Klasse aufgerufen werden
  - werden über den Klassennamen aufgerufen
  - dürfen nur auf Klassenvariablen zugreifen
  - Modifikator static vor der Methoden-Deklaration

Destruktor

- trägt den gleichen Namen wie die Klasse, mit ~ (Tilde) davor
- wenn kein eigener Destruktor definiert wird, dann stellt der Compiler einen Standard-Destruktor bereit
- typischer Einsatz, wenn
  - dynamisch reservierter Speicher freigegeben werden soll
  - Dateien geschlossen und Datei-Handles freigegeben werden sollen
- wird automatisch aufgerufen, kurz bevor ein Objekt seine Gültigkeit verliert (unmittelbar vor der Zerstörung)
  - Stack: wenn der Block (Scope) verlassen wird
  - Heap: wenn delete aufgerufen wird

Beispiel

```
class Point {
    ~Point() {
        cout << "Destruktoren der Attribute werden automatisch aufgerufen" << endl;
    }
}
```

Initialisierung- und Zerstörungsreihenfolge

- ctor initialisiert Attribute in Deklarations-Reihenfolge
  - danach folgt eigener Block
- dtor führt zuerst eigenen Block aus
  - und zerstört danach die Attribute in umgekehrter Reihenfolge
- Beispiel

```
struct A {  };
struct B {  };
struct C { A m_a; };
struct D { A m_a; B m_b; C m_c; };

{
      D d;     // ctor A, ctor B, ctor A, ctor C, ctor D
}              // dtor D, dtor C, dtor A, dtor B, dtor A
```

Kopierkonstruktor

- wird zum Kopieren eines Objektes verwendet (flache oder tiefe Kopie)
- verwendet genau einen Parameter: const-Referenz auf Objekt derselben Klasse
- eigener Kopierkonstruktor für tiefe Kopien implementieren
- wird ein eigener Kopierkonstruktor angeboten, so sollte auch ein eigener und kompatibler Zuweisungsoperator angeboten werden

Beispiel

```
Point(const Point& p)
    : m_x(p.m_x), m_y(p.m_y), m_z(p.m_z), m_color(p.m_color) {
}
```

Anwendungen

```
Point p1(1, 2, 3, 4);
Point p2(p1);                       // expliziter Aufruf des Kopierkonstruktors
Point p3 = p1;                      // impliziter Aufruf des Kopierkonstruktors
Point p5; p5 = Point(2, 3, 4, 5);   // schlecht: 1. Standardkonstruktor
                                    //           2. benutzerdef. Konstruktor
                                    //           3. Zuweisungsoperator
```

RAII: Resource Allocation is Initialization

- Grundsätze
  - beim Erzeugen eines Objekts (einer Ressource) muss das Objekt vollständig initialisiert werden → Aufgabe des Konstruktors
  - beim ordentlichen Verlassen des Konstruktors immer ein gültiges Objekt zurücklassen
  - im Fehlerfall sollte der Konstruktor mit einer Exception beendet werden, das bedeutet, dass bereits angeforderte Ressourcen wieder freigegeben werden müssen
- problematisches Beispiel: kann zu memory leak führen

```
struct StereoImage {
    Image *left, *right;    // was passiert, wenn der Heap nur für left reicht?
    StereoImage() : left(new Image), right(new Image) {}
    ~StereoImage() { delete left; delete right; }
};
```

Default-Methoden (1)

- Idee
  - Klassen haben eine Reihe von Konstruktoren und Methoden, die der Compiler automatisch bei Bedarf generiert (synthetisiert), falls diese Konstruktoren/Methoden nicht benutzerdefiniert werden.
- Standard-Konstruktor C::C()
  - nur wenn kein benutzerdefinierter Konstruktor erstellt wird
- Destruktor C::~C()
  - nur wenn kein benutzerdefinierter Destruktor erstellt wird
- Kopieroperationen (flache Kopie)
  - nur wenn keine eigenen Kopier- oder Verschiebeoperationen definiert worden sind und wenn sich alle Attribute kopieren lassen
  - Kopierkonstruktor C::C(const C&)
  - Zuweisungsoperator C& operator=(const C&)

Default-Methoden (2)

- Verschiebeoperationen
  - nur wenn keine Kopieroperationen und kein Destruktor definiert worden sind
  - wird ein eigener Verschiebekonstruktor angeboten, so sollte auch der Verschiebeoperator implementiert werden
  - Verschiebekonstruktor C::C(C&&)
  - Verschiebeoperator C& operator=(C&&)

C++ Ausdrücke

- C++-Ausdrücke können anhand zwei unabhängiger Eigenschaften charakterisiert werden
  - Datentyp
  - Wertekategorie
- jeder C++-Ausdruck
  - hat einen Nicht-Referenzdatentyp
  - und gehört einer von drei Wertekategorien an

Wertekategorien

- 2 Unterscheidungsmerkmale
  - Ausdruck hat eine Identität
  - Wert kann verschoben werden
- Primäre Wertekategorien
  - nur 3 der 4 möglichen Kombinationen werden in C++ verwendet
  - typischer x-value: std::move(x)
- Sekundäre Wertekategorien
  - gl-value (general left value) = l-value ∪ x-value
    - hat Identität
  - r-value (right value) = x-value ∪ pr-value
    - kann verschoben werden
    - Adressoperator kann nicht verwendet werden

	keine Identität	hat Identität
kann nicht verschoben werden		l-value
kann verschoben werden	pr-value	x-value

gl-value

r-value

Wertekategorien: Code-Beispiel

```
void test(int& x)  {
   cout << "non movable" << endl;
}
void test(int&& x) {
   cout << "movable" << endl;
}
int main() {
   int x = 5; cout << &x << endl;
   test(x);
   test(5);
   {
      int x = 5; cout << &x << endl;
   }
}
```

Ausdrücke und Wertekategorien

- l-value (left value): hat Identität, nicht verschiebbar
  - Variable, Funktion, Klassenattribut, …
  - Parameter, auch wenn vom Typ r-value Referenz
  - Funktionsaufruf mit Rückgabetyp l-value Referenz
  - Stringliteral
- pr-value (pure right value): keine Identität, verschiebbar
  - Literal: 42, true, nullptr, …
  - arithmetischer Ausdruck: a + b, a < b, …
  - Funktionsaufruf: str.substr(1, 2), …
- x-value (expiring value): hat Identität, verschiebbar
  - Funktionsaufruf mit Rückgabetyp r-value Referenz
  - Array- oder Attributzugriff bei einem r-value

Temporäre Objekte (r-value)

- 2 Arten der Lebensverlängerung sind möglich

```
int main() {
    std::string s1 = "Test";
    std::string&& r1 = s1;          // r-value Referenz darf nicht zu l-value binden
    std::string& r1 = s1 + s1;      // l-value Referenz darf nicht zu r-value binden

    // konstante l-value Referenz darf zu temporärem Objekt (s1 + s1) binden
    const std::string& r2 = s1 + s1; // verlängert die Lebensdauer des temp. Obj.
    // r2 += "Test"; // von r2 gebundener const String darf nicht verändert werden

    // r-value Referenz bindet zu temporärem Objekt (s1 + s1)
    std::string&& r3 = s1 + s1;     // verlängert die Lebensdauer des temp. Objekts
    r3 += "Test";                   // erlaubt, weil r3 eine r-value Referenz ist
    std::cout << r3 << '\n';
}
```

Parameter vom Typ r-value Referenz

- Funktionsparameter sind innerhalb der Funktion l-values

```
string foo(string&& s) {
    s += "456";                    // innerhalb von foo ist s ein lvalue
    return move(s);                // stellt sicher, dass Move-Semantik
}                                  // für die Rückgabe verwendet wird
…
zur Erzeugung von t wird der Verschiebekonstruktor verwendet
string t = foo(string("123"));     // foo wird mit einem temporären
                                   // String-Objekt aufgerufen

string x;
foo(x);          // ein l-value darf nicht an foo übergeben werden
```

Besitzübernahme (Move-Semantik)

- Idee der Besitzübernahme
  - temporäre Objekte werden kurz nach der Erstellung wieder zerstört
  - werden dem temporären Objekt vor seiner Zerstörung die Daten entzogen, so stört das nicht weiter

```
class PointVector {
    unique_ptr<Point[]> m_array;
    size_t m_size;
public:
    PointVector(size_t s = 0) …
    void add(const Point& p) { … }
};
PointVector create() {
    PointVector v;
    v.add(Point(1,2,3));
    return v;
}
```

```
int main() {
    // Verschiebekonstruktor
    PointVector pv1 = create();
    PointVector pv2(create());

    // Verschiebeoperator
    PointVector pv3;
    pv3 = create();
}
```

Daten des PointVector aus create() an das neue Objekt übertragen!

Umsetzung der Move-Semantik

```
class PointVector {
    unique_ptr<Point[]> m_array;
    size_t m_size;
public:
    // benötigt eigenen Standardkonstruktor und Destruktor
    // Verschiebekonstruktor
    PointVector(PointVector&& v) : m_array(std::move(v.m_array)), m_size(v.m_size) {
        v.m_size = 0;
    }

    // Verschiebeoperator
    PointVector& operator=(PointVector && v) {
        if (this != &v) {
            m_size = v.m_size; v.m_size = 0;
            m_array = std::move(v.m_array);  // unique_ptr hat keinen Zuweisungsoperator
        }                                    // aber einen Verschiebeoperator
        return *this;
    }
};
```

std::exchange und std::swap

- `T exchange( T& obj, U&& new_value );`
  - ersetzt den Wert von obj mit dem neuen Wert new_value und gibt den alten Wert von obj zurück
  - eignet sich gut für die Implementierung des Verschiebekonstruktors

```
PointVector(PointVector&& v)
    : m_array(std::exchange(v.m_array, nullptr))
    , m_size(std::exchange(v.m_size, 0))
{}
```

- `void swap( T& a, T& b );`
  - vertauscht die Werte der beiden Variablen a und b
  - eignet sich gut für die Implementierung des Verschiebeoperators

```
PointVector& operator=(PointVector && v) {
    if (this != &v) {
        std::swap(m_size, v.m_size);
        std::swap(m_array, v.m_array);
    }
    return *this;
}
```

std::move

- std::move(T x)
  - ist im Wesentlichen ein Typkonvertierungsoperator, um aus x eine rvalue Referenz zu machen: static_cast<T&&>(x)
  - verschiebt selber gar nichts
  - stellt sicher, dass der Compiler einen allfälligen Verschiebekonstruktor bzw. Verschiebeoperator anstatt dem Kopierkonstruktor bzw. Zuweisungsoperator aufruft
- Einsatzzweck
  - Aufruf des Verschiebekonstruktors/-operators erzwingen
- Beispiel

```
std::string s1 = "hello";
std::string s2 = std::move(s1);   // Verschiebekonstruktor
// s1 == ""                       // hinterlässt in s1 gültiges Objekt
// s2 == "hello"                  // aber die Daten sind nun in s2
```

Überladen von Methoden

- Signatur einer Methode besteht aus
  - Namensraum, Klasse, Name, Parameterliste
    - Anzahl und Typen der Parameter (Parameterbezeichner sind irrelevant)
  - Rückgabetyp gehört nicht dazu
- alle Methoden müssen eine eindeutige Signatur haben
- Überladen von Methoden
  - wenn mehrere Methoden im selben Namensraum bzw. Klasse denselben Namen, aber dennoch nicht die gleiche Signatur haben
- Beispiel

```
class Point {
    double m_x, m_y, m_z;
public:
    Point& move(double x, double y = 0, double z = 0);
    Point& move(double delta[3]);
    Point& move(const Point& p);
};
```

Operatoren überladen

- Idee
  - nicht nur Methoden sondern auch Operatoren können überladen werden (bekanntes Beispiel: << für die Ausgabe)
  - ermöglicht schönere Syntax (infix anstatt präfix) als mit Methoden

```
Complex c1(2, 4), c2(2, -4);
Complex c = c1 + c2/10;
```

- Grundregeln
  - es können keine neuen Operatoren definiert werden
  - vorgegebene Vorrangregeln dürfen nicht verletzt werden
  - Überladen von && und || deaktiviert short-circuit-Evaluierung
  - mindestens ein Argument des Operators muss ein Objekt sein oder der Operator muss eine Instanzmethode sein
    → damit wird verhindert, dass die Operatoren der primitiven Datentypen verändert werden

Operatoren

- überladen erlaubt für

new	+	~	>	/=	|=	<<=	>=	++	->	%
delete	-	^	!	+=	%=	<<	==	--	()	[]
new[]	*	&	=	-=	^=	>>	!=	&&	->*	,
delete[]	/	|	<	*=	&=	>>=	<=	||	""	

- ab C++20
  - <=> *spaceship operator* entspricht dem compareTo aus Java
  - co_await gibt in Coroutine die Kontrolle an Aufrufer zurück

- überladen nicht erlaubt für

.	.*	::	? :

- mehr Details dazu in Wikipedia

Operator als Funktionsaufruf

Element-Funktion	Syntax	Ersetzung durch
nein	`x ⊗ y`	`operator⊗(x,y)`
	`⊗ x`	`operator⊗(x)`
	`x ⊗`	`operator⊗(x,0)`
ja	`x ⊗ y`	`x.operator⊗(y)`
	`⊗ x`	`x.operator⊗()`
	`x ⊗`	`x.operator⊗(0)`
	`x = y`	`x.operator=(y)`
	`x(y)`	`x.operator()(y)`
	`x[y]`	`x.operator[](y)`
	`x->`	`(x.operator->())->`
	`(T)x`	`x.operator T()`

`T` ist Platzhalter für einen Datentyp

friend-Methoden

- Operatoren und Methoden können als freie Funktionen implementiert werden
  - haben keinen versteckten this-Parameter
  - haben standardmässig nur Zugriff auf öffentliche Attribute der Parameter
  - mittels friend kann der Zugriff auf alle Attribute erweitert werden
  - sollten primär für symmetrische Operatoren verwendet werden
  - ermöglicht dem Compiler mehr implizite Konvertierungen
- Beispiel

```
class Point {
public:
    friend bool operator<(const Point& lhs, const Point& rhs);
};

bool operator<(const Point& lhs, const Point& rhs) {
    return lhs.m_x < rhs.m_x || …;
}
```

Spaceship Operator <=>

- Semantik
  - ähnlich zu Java's compareTo, aber Rückgabetyp ist entweder
    - auto oder
    - einer der nachfolgenden drei Vergleichsklassentypen

Rückgabetyp	als gleich bewertete Werte sind …	inkompatible Werte sind …
std::strong_ordering	ununterscheidbar	nicht erlaubt
std::weak_ordering	unterscheidbar	nicht erlaubt
std::partial_ordering	unterscheidbar	erlaubt

  - strong_ordering
    - equal, less, greater
  - weak_ordering
    - equivalent, less, greater
  - partial_ordering
    - equivalent, less, greater, unordered

Realisierung der Klasse Person

```
// in h-Datei
class Person {
    string m_name;              // Aggregation: Person hat einen Namen
    int m_age;                  // Aggregation: Person hat ein Alter

public:
    Person(const char name[], int age) : m_name(name), m_age(age) {}
    string getName() const   { return m_name; }
    void setAge(int age)     { m_age = age; }
    void print() const;         // keine inline-Implementierung
};

// in cpp-Datei
void Person::print() const {
    cout << "Name: " << m_name << endl;
    cout << "Alter: "   << m_age << endl;
}
```

Realisierung der Klasse Student

```
// in h-Datei: Vererbung: ein Student ist eine Person
class Student : public Person {
    // die Klasse Student wird von der Klasse Person abgeleitet
    // und erbt alle Attribute und Methoden der Klasse Person
    int m_number;

public:
    Student(const string& name, int age, int nr)
        : Person(name, age), m_number(nr) {}
    // neue Methoden der Klasse Student
    void setNumber(int nr) { m_number = nr; }
    void printNumber() const;
};

// in cpp-Datei
void Student::printNumber() const {
    cout << "Studentennummer: " << m_number << endl;
}
```

Verwendung der Klasse Student

```
void main () {
    Person pers("Peter", 20);
    pers.setAge(21);
    pers.print();

    Student student("Anna", 21, 50101);
    student.setName("Anne");
    student.setNumber(56123);
    student.print();                    // gibt keine Studentennummer aus
    student.printNumber();              // gibt Studentennummer aus

    Person pers2 = student;             // Projektion von Student auf Person (Kopie)
    pers2.print();                      // gibt keine Studentennummer aus
}
```

Konstruktoren in abgeleiteten Klassen

- Idee
  - jeder abgeleitete Konstruktor initialisiert nur die neuen Attribute
  - vererbte Attribute werden vom Konstruktor der Basisklasse initialisiert
- Umsetzung
  - In der Initialisierungsliste des Konstruktors wird der Konstruktor der Basisklasse aufgerufen
  - falls kein expliziter Aufruf eines Konstruktors der Basisklasse erfolgt, wird der Standardkonstruktor der Basisklasse implizit aufgerufen
  - Aufgaben der Initialisierungsliste (Reihenfolge beachten)
    - Aufrufen von Konstruktoren der Basisklasse(n)
    - Aufrufen von anderen Konstruktoren der eigenen Klasse (Constructor delegation) oder Initialisieren der eigenen Attribute

Destruktor einer abgeleiteten Klasse

- Konzept
  - der Destruktor einer abgeleiteten Klasse ruft nach Ausführung seines Methodenkörpers den Destruktor der Basisklasse implizit auf
  - dynamische Attribute können im Destruktor zuerst gelöscht werden, bevor Attribute der Basisklasse gelöscht werden
- Wann soll ein Destruktor ausprogrammiert werden?
  - wenn die Klasse Attribute enthält, welche eigenständig mit new erzeugt worden sind, so müssen diese im Destruktor wieder gelöscht werden
- Wird der Destruktor auch bei einem statisch erzeugten Objekt aufgerufen?
  - Ja! Beim Verlassen des Blocks, in dem das Objekt erstellt worden ist, wird zuerst der Destruktor aufgerufen, bevor das Objekt vom Stack entfernt wird.

Typkonvertierungen von Zeigern

- Typ einer Zeiger- oder Referenzvariable muss nicht gleich dem Typ des Objektes sein, auf welches die Zeiger-/Referenzvariable verweist
  - bisher: `Student *pStud = new Student("Anna", 21, 50101);`
  - neu: `Person *pPers = new Student("Anna", 21, 50101);`

- implizite (automatische) Zeigertypkonvertierung (Up-Cast)

```
Person *pPers2 = pStud;                                  // impliziter Up-Cast
```

- explizite Zeigertypkonvertierung (Down-Cast)

```
Student *pStud2 = dynamic_cast<Student*>(pPers); // expliziter Down-Cast
```

Gültige Up- und Down-Casts

- Up-Cast
  - Konvertierung in einen Zieltyp, der in der Vererbungshierarchie weiter oben liegt
  - implizite Konvertierung
  - immer gültig, wenn der Zieltyp ein Vorfahre ist
- Down-Cast
  - Konvertierung in einen Zieltyp, der in der Vererbungshierarchie weiter unten liegt
  - nur explizite Konvertierung möglich
  - nur gültig, wenn der Zeiger auf ein Objekt des Zieltyps oder einer abgeleiteten Klasse des Zieltyps zeigt
- Beispiele

```
Person *pPers = new Person();
Student *pStud = new Student();
Person *pS = pStud;                          // impliziter Up-Cast
Student *pS2 = static_cast<Student*>(pS);    // gültiger Down-Cast
Student *pS3 = static_cast<Student*>(pPers); // ungültiger Down-Cast
```

Runtime Type Information (RTTI)

- Problem
  - static_cast oder C-Cast führen bei ungültigem Down-Cast zu Laufzeitfehlern
- RTTI
  - speichert genauen Typ zu jeder Instanz
  - kann bei Bedarf abgeschaltet werden
- dynamic_cast
  - bei einem gültigen Down-Cast
    - funktioniert wie ein static_cast
    - `Student *pS4 = dynamic_cast<Student*>(pS);` `// pS4 == pS`
  - bei einem ungültigen Down-Cast
    - gibt einen nullptr zurück (bei einer Zeigervariablen)
    - `Student *pS5 = dynamic_cast<Student*>(pPers);` `// pS5 == nullptr`
    - wirft bad_cast Exception (bei einer Referenzvariablen)

Typkonvertierung mit Smart-Pointers

- Funktioniert analog zu Zeigern

```
shared_ptr<Person> spP = make_shared<Person>();
shared_ptr<Person> spS = make_shared<Student>();
```

- gültige Down-Casts

```
auto sp1 = static_pointer_cast<Student>(spS);
auto sp2 = dynamic_pointer_cast<Student>(spS);
```

- ungültiger Down-Cast

```
auto sp3 = dynamic_pointer_cast<Student>(spP); // sp3 == nullptr
```

Polymorphie (Vielgestaltigkeit)

- Polymorphie von Operationen
  - gleiche Methodenaufrufe in verschiedenen Klassen führen zu klassenspezifischen Anweisungsfolgen
  - Beispiel: pPers->print() vs. pStud->print()

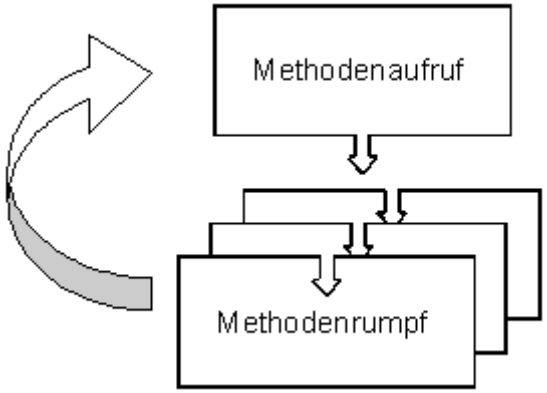
- Polymorphie von Objekten (nur bei Vererbungshierarchien)
  - an die Stelle eines Objektes in einem Programm kann auch ein Objekt einer abgeleiteten Klasse treten
  - ein abgeleitetes Objekt ist polymorph: es kann sich auch als Objekt einer Basisklasse ausgeben
  - Beispiel: ein Student verhält sich wie ein Student, kann sich aber auch wie eine Person verhalten

Statische und dynamische Bindung

- Bindung

  Zuordnung eines Methodenrumpfes zum Aufruf einer Methode

- statische (frühe) Bindung
  - Zuordnung erfolgt zur Kompilationszeit
  - erlaubt Methodenaufrufe durch Methodencode zu ersetzen
  - Standardverhalten
- dynamische (späte) Bindung
  - Zuordnung erfolgt erst zur Laufzeit des Programms
  - sehr mächtiges Konzept, weil es die Wiederverwendung von Programmcode drastisch erhöht
  - muss explizit mit dem Schlüsselwort virtual deklariert werden
  - benötigt pro Objekt einen versteckten Zeiger auf eine Tabelle (vtable) mit den dynamisch gebundenen Methoden



Überschreiben von Methoden

- Idee
  - in einer abgeleiteten Klasse kann eine Methode überschrieben (override) werden
  - die überschriebene Methode hat
    - die gleiche Signatur (Name und Parameterliste)
    - und den gleichen Rückgabetyp oder bei Referenz-/Zeigertyp auch eine Spezialisierung davon
  - wird eine Methode in einer Basisklasse als virtual deklariert, so sind auch alle überschriebenen Methoden davon virtual
- Beispiel

```
class Person { … virtual void print() const { … } … };
class Student : public Person {
    void print() const override {    // die Methode soll explizit als
        …                            // überschrieben gekennzeichnet werden
        Person::print();   // Aufruf der Methode print() aus der Basisklasse
    }
};
```

Gebundene Methoden

- Falls Methoden nicht virtual sind: statische Bindung
  - der statische Typ des Objekts, Zeigers oder Referenz entscheidet über die Wahl der aufgerufenen Methode
- Falls Methoden virtual sind: dynamische Bindung
  - Zugriff über Zeiger/Referenz: Polymorphie kommt zum Einsatz
    - und der dynamische Typ des Zeigers oder der Referenz entscheidet über die Wahl der aufgerufenen Methode
  - direkter Zugriff: Polymorphie kommt nicht zum Einsatz
    - weil die Methode nicht über einen Zeiger bzw. Referenz aufgerufen wird
  - Beispiele

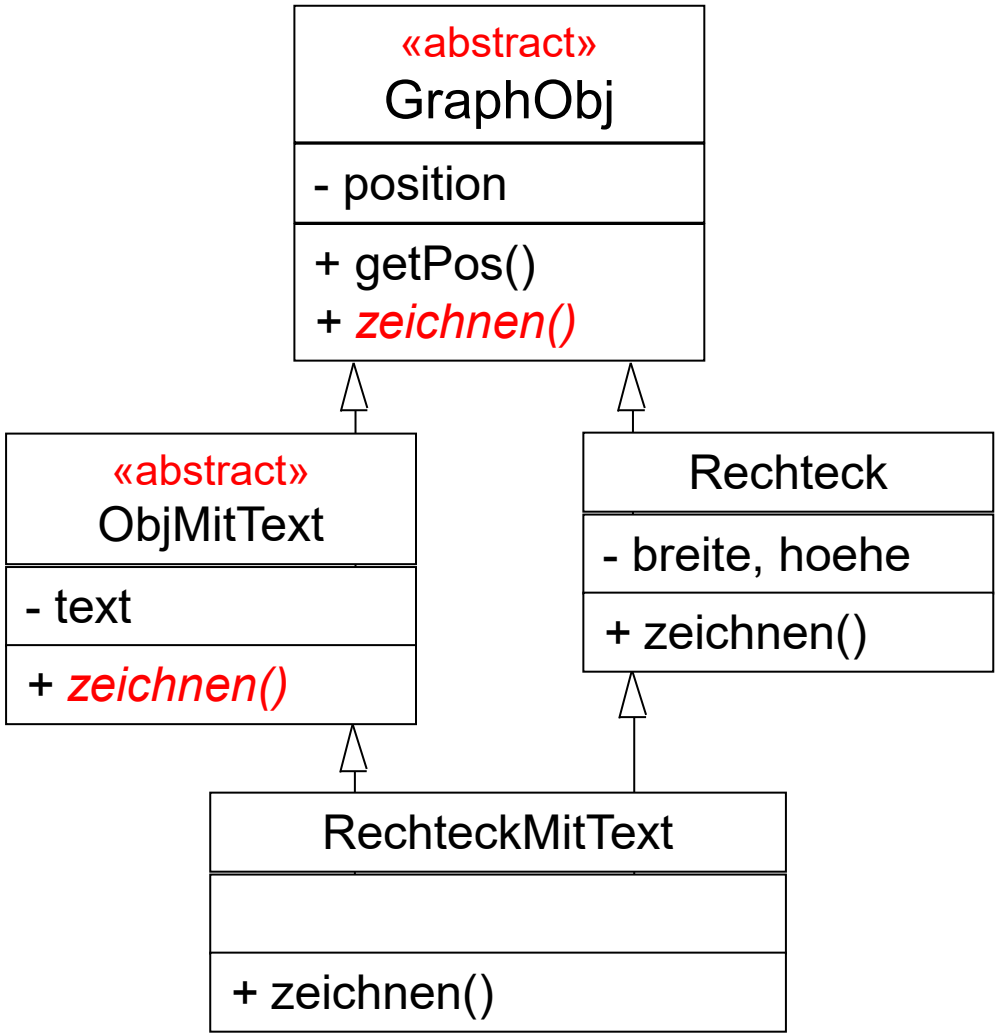
```
Person p, *pP;
Student s, *pS = new Student();
p = s; p.print();              // print() der Klasse Person wird aufgerufen
pP = pS; pP->print();          // print() der Klasse Student wird aufgerufen
Person& rP = s; rP.print();    // print() der Klasse Student wird aufgerufen
```

Destruktoren

- Bei shared_ptr geschieht das Richtige automatisch, d.h. der Basisklassen-Destruktor muss nicht virtuell sein
  - das Ref-Counter-Objekt kennt nur den dynamischen Typ und ruft daher den richtigen Destruktor auf
- Beim Einsatz von unique_ptr sollte der Basisklassendestruktor virtuell sein.
- Achtung!
  - sobald wir virtual ~C() = default; deklarieren, verlieren wir den Verschiebekonstruktor und den Verschiebeoperator. Daher …
- «Rule of Zero» und «Rule of 5 Defaults»
  - wenn nicht nötig, definieren wir keine der fünf Spezialfunktionen (default-ctor, copy-ctor, move-ctor, assignment-op, move-op) und lassen den Compiler diese automatisch generieren
  - wenn wir einen virtuellen Destruktor benötigen, dann definieren wir gleich alle fünf Spezialfunktionen als default, damit wir die Verschiebefunktionen nicht verlieren

Mehrfachvererbung

- Beispiel aus der Welt der grafischen Objekte
- hier mit gemeinsamer Basisklasse
  (ist nicht notwendig)
- Probleme: Namenskonflikte, Mehrdeutigkeiten
- meistens nur für Interfaces sinnvoll



Probleme der Mehrfachvererbung

- Beispiel

```
Rechteck r(0, 0, 20, 50);
RechteckMitText br(10, 5, 60, 60, "Text");
r.zeichnen();  // ruft zeichnen() von Rechteck auf
br.zeichnen() // ruft zeichnen() von RechteckMitText auf

Position rPos = r.getPos();       // gibt Ursprung des Rechtecks zurück
Position brPos = br.getPos();     // → Compiler-Fehler
GraphObj *pObj = &br;             // → Compiler-Fehler
```

- Warum ein Compiler-Fehler?
  - br.getPos() ist nicht eindeutig, denn es könnte getPos() von ObjMitText oder von Reckteck aufgerufen werden
  - Ursache: Teilobjekt GraphObj ist zweimal vorhanden und nicht beide Teilobjekte müssen identisch sein, d.h. die gleiche position besitzen

C++ Templates (Generics)

- Generische Klassen und Funktionen in C++
  - Template = Schablone = parametrisierbarer Typ
  - typsichere Funktionen und Klassen (Makros machen nur Textersetzung)
  - Quellcode vereinfachen, flexibilisieren und in der Länge reduzieren
  - Implementierung direkt in Header-Dateien
  - erhöht die Kompilationszeit
- Beispiel für überladene Funktion

```
int min( int a, int b ) {
        return ( a < b ) ? a : b; // min for ints
}
char min( char a, char b ) {
        return ( a < b ) ? a : b; // min for chars
}
```

- jetzt generisch

```
template<typename T>
T min( T a, T b ) {
        return ( a < b ) ? a : b; // generic min
}
```

```
// Einsatz
int main() {
      int m = min(5, 7);
}
```

Instanziierung

- Instanziierung
  - durch Instanziierung wird aus einem Template eine vollständige Funktion oder Klasse
  - Template wird bei Verwendung implizit instanziiert
  - erfolgt in jeder Kompilationseinheit von Neuem
  - durch explizite Instanziierung kann die Kompilationszeit verkürzt werden

- Beispiel einer generischen Klasse

```
template <typename T>
class A {
    T m_t;
public:
    A(T t) : m_t(t) {}
    void set(T t) { m_t = t; }
};
```

```
// Implizite Instanziierung
int main() {
    A<double> a(3.5);
}
```

Funktions-Template

- Syntax

  template < *TemplateParamListe* > *Funktionsdefinition*

- mit
  - *TemplateParamListe*
    - kommaseparierte Liste von Parametern, welche Typparameter oder Wertparameter sein können; Default-Werte sind erlaubt
    - Beispiele

```
typename TypBezeichner                  // für beliebigen Datentyp (auch primitiv)
class TypBezeichner                     // für beliebige Klasse
template < TemplateParamListe >         // nicht instanziierter generischer Typ
IntegralTypBezeichner Variable          // Wert-Parameter
```

  - *TypBezeichner*
    - ein beliebiger Name, der in der Funktionsdefinition als Datentyp verwendet wird
    - sowohl Grunddatentypen als auch Klassen sind möglich
  - *Funktionsdefinition*
    - übliche Funktionsdefinition, Methode, Konstruktor
    - darf auch ein überladener Operator sein

Klassen-Templates

- Syntax

  template < *TemplateParamListe* > *Klassendefinition*

- mit
  - *TemplateParamListe:* wie bei Funktions-Templates
- Beispiele von generischen Klassen

```
template<typename T>
class Vector {
    T* m_array;
    …
};
template<typename T = char>      // default Zeichentyp ist char
class String {
    T* m_string;
};
```

Templates mit Wert-Parameter

- Wert-Parameter
  - ganzzahlig (ab C++20 auch floating-point möglich)
- Beispiel: statisches Array mit variabler Länge

```
template<typename T, size_t S>
class Array {
    T m_array[S];
public:
    const T& operator[ ](size_t pos) const    { return m_array[pos]; }
    T& operator[ ](size_t pos)                { return m_array[pos]; }
    void print() const {
        cout << '[';
        cout << m_array[0];
        for(size_t i = 1; i < S; i++) cout << ',' << m_array[i];
        cout << ']' << endl;
    }
};
```

Spezialisierung von Templates

- Beispiel
  - Minimum von zwei Zahlen oder Zeichen bestimmen
  - bei Zeichen soll die Gross-/Kleinschreibung nicht beachtet werden
- Allgemeinfall

```
template<typename T> T min( T a, T b ) {
        return ( a < b ) ? a : b; // generic min
}
```

- Spezialisierung (muss nach dem Allgemeinfall folgen)

```
template<> char min<char>(char a, char b) {
    a = tolower(a);
    b = tolower(b);
    return ( a < b ) ? a : b;
}
```

Auflistung aller nicht spezialisierten Template-Parameter

Variadic Templates

- Templates mit beliebiger Anzahl Argumente
  - Einsatz von Parameter Packs (…)
  - Pattern-Matching zur Kompilationszeit
- Einsatz bei Klassen

```
template<typename… Ts> class C { };
```

Bestimmung der Anzahl Parameter innerhalb der Klasse C

```
size_t types = sizeof…(Ts);
```

- Einsatz bei Funktionen

```
template<typename… Ts> void func(const Ts&… vs) { }
```

Bestimmung der Anzahl Parameter innerhalb der Funktion func

```
size_t params = sizeof…(vs);
```

Abhängige Typnamen

- Zweck
  - mit dem Schlüsselwort typename kann dem Compiler mitgeteilt werden, dass ein unbekannter Bezeichner ein Typ ist
  - typename muss verwendet werden, wenn der unbekannte Bezeichner ein vom Template abhängiger qualifizierter Name ist

- Beispiel

```
template<typename T>
struct Extrema {
   using type = typename T::value_type;
   type m_min, m_max;
   Extrema(const T& data)
   : m_min(*min_element(begin(data), end(data)))
   , m_max(*max_element(begin(data), end(data))) {}
};
Extrema<vector<int>> x({ 8, 3, 5, 6, 1, 3 });
```

Constraints und Concepts

- requires *requires_expression*
  - verschiedenste Arten von Bedingungen werden unterstützt
- concept
  - eine logische Verknüpfung von Bedingungen (constraints)

```
template<class T, class U>
concept Same = is_same_v<T, U> && is_same_v<U, T>;
```

- Anwendung (verschiedene Schreibweisen möglich)

```
template<Same<int> T>
void foo(T* p) {
        cout << "foo" << endl;
}
```

Verwendung von Concepts

- Konzepte sind sinnvoll für Template-Parameter und für nicht definierte Parameter/Rückgabetypen (auto)

```
void funcWithAutoInline(const std::convertible_to<std::string> auto& x) {
    std::string v = x;
}
template <std::convertible_to<std::string> T>
void funcWithTemplateInline(const T& x) {
    std::string v = x;
}
template <typename T> requires std::convertible_to<T, std::string>
void funcWithTemplatePostfix(const T& x) {
    std::string v = x;
}
```

Standardeingabe und -ausgabe in C++

- Standardeingabe
  - Lesen eines Bytestroms von der Tastatur
  - Verwendung eines Objekts der Klasse istream (z.B. cin)
- Standardausgabe
  - Schreiben eines Bytestroms auf den Bildschirm
  - Verwendung eines Objekts der Klasse ostream
    - cout: Standardausgabe
    - cerr: Standardfehlerausgabe
    - clog: gepufferte Standardfehlerausgabe
- Beispiel

```
struct S { string m_name; uint32_t m_flags; double m_value; } s;
cin >> s.m_name >> s.m_flags >> s.m_value;
cout << "s: name = " << s.m_name << ", flags = " << s.m_flags
     << ", value = " << s.m_value << endl;
```

Datenströme (Streams)

- Was ist ein Datenstrom?
  - geordnete Folge von Datenbytes mit unbekannter Länge (Anzahl von Bytes)
- Eingabestrom (input stream)
  - Datenstrom, der aus einer Datenquelle kommt
  - Beispiel: Zeichen, die über die Tastatur eingegeben werden
- Ausgabestrom (output stream)
  - Datenstrom, der zur einer Datensenke gesendet wird
  - Beispiel: Zeichen, die auf den Bildschirm geschrieben werden

- Wo finde ich Infos dazu?
  - Streams sind Teil der Standard-Bibliothek
  - C++ Standard library

Ein- und Ausgabe

- Formatierte Ein- und Ausgabe
  - Ausgabe: bei der formatierten Ausgabe wird ein Wert/Objekt als Zeichenkette in einen Ausgabestrom geschrieben
    - es wird der operator<<(…) verwendet
  - Eingabe: bei der formatierten Eingabe wird eine Zeichenkette aus einem Eingabestrom gelesen, die Zeichenkette geparst und ein Wert/Objekt des gewünschten Datentyps mit Daten abgefüllt
    - es wird der operator>>(…) verwendet
    - falls der Parser einen Fehler feststellt, wird der Wert/Objekt nicht abgefüllt und der Eingabestrom wird in einen Fehlerzustand (failbit) gesetzt
- Unformatierte Ein- und Ausgabe
  - Ausgabe: Daten werden mit write(…) als Zeichenfolge in den Ausgabedatenstrom geschrieben
  - Eingabe: Daten werden mit read(…) als Zeichenfolge aus dem Eingabestrom gelesen

Formatierte Ein- und Ausgabe

```
class Person {
    std::string m_name;
    std::string m_givenName;
    int m_age;
    bool m_female;

public:
    Person(…) {}

    friend std::ostream& operator<<(std::ostream& os, const Person& p) {
        return os << p.m_givenName << " " << p.m_name << std::boolalpha
            << (p.m_female ? " (female)" : " (male)") << " is "
            << p.m_age << " years old";
    }

    friend std::istream& operator>>(std::istream& is, Person& p) {
        return is >> p.m_givenName >> p.m_name >> p.m_age
            >> std::boolalpha >> p.m_female;
    }
};
```

Zustände von Datenströmen

- Zustand eines Datenstromes ist eine Zahl, `iostate`, welche mit `rdstate()` ausgelesen werden kann:
  - 0 bedeutet, dass alles in Ordnung ist.
  - alle anderen Zahlen bedeuten, dass sich der Strom in einem Fehlerzustand befindet, wobei eines oder mehrere Fehlerbits gesetzt sind
- Abfragen einzelner Bits von `iostate` mit
  - `good()`
  - `eof()`
  - `fail()`
  - `bad()`
- Manuelles Setzen des fail Bits von `iostate` mit
  - `setstate(std::ios::failbit)`

Stream-Manipulatoren

- Idee
  - anstatt dem mühsamen Setzen von Flags (z.B. mit setf(..))
    werden Stream-Manipulatoren gezielt in den Datenfluss integriert
- einfaches Beispiel
  - vorher

```
cout.setf(ios::hex, ios::basefield);      // Zahlenbasis auf 16 setzen
cout.setf(ios::uppercase);                // Hexzahlen mit Grossbuchstaben
cout << i << endl;
```

  - nachher

```
cout << hex << uppercase << i << endl;
```

- weiteres Beispiel
  - vorher

```
cout.width(3);                            // Zahlenbreite: 3
cout.fill('0');                           // mit füllenden Nullen auffüllen
cout << i << endl;
```

  - nachher

```
cout << setw(3) << setfill('0') << i << endl;
```

- was steckt dahinter?

Unformatierte Ein- und Ausgabe

- Unformatierte Eingabe
  - peek gibt Vorschau auf das nächste Zeichen im Zeichenstrom
  - get liest ein Zeichen vom Zeichenstrom
  - read liest *n* Zeichen vom Zeichenstrom
  - getline liest eine ganze Zeile oder bis zu einem angegebenen Trennzeichen

    (beim Übergang von formatierter zu unformatierter Eingabe können mit is >> ws nicht konsumierte Whitespaces vorgänging konsumiert werden)
  - ignore überliest und ignoriert Zeichen im Zeichenstrom
  - gcount gibt Anzahl verarbeitete Zeichen der letzten unformatierten Eingabe zurück
  - unget macht das zuletzt gelesene Zeichen im Zeichenstrom nochmals verfügbar
- Unformatierte Ausgabe
  - put schreibt ein Zeichen in den Zeichenstrom
  - write schreibt *n* Zeichen in den Zeichenstrom

C++ Standardbibliothek (1)

Bibliothek	C++-Headers
Algorithms	<algorithm> <execution>
Atomic Operations	<atomic>
C Compatibility	<cassert> <cctype> <cerrno> <cfenv> <cfloat> <cinttypes> <climits> <clocale> <cmath> <csetjmp> <csignal> <cstdarg> <cstddef> <cstdint> <cstdio> <cstdlib> <cstring> <ctime> <cuchar> <cwchar> <cwctype>
Concepts	<concepts>
Containers	<array> <deque> <forward_list> <list> <map> <queue> <set> <stack> <unordered_map> <unordered_set> <vector>
Coroutines	<coroutine>
Filesystem	<filesystem>
Input/Output	<fstream> <iomanip> <ios> <iosfwd> <iostream> <istream> <ostream> <sstream> <streambuf> <syncstream>

C++ Standardbibliothek (2)

Bibliothek	C++-Headers
Iterators	<iterator>
Localization	<locale>
Numerics	<bit> <complex> <numbers> <numeric> <random> <ratio> <valarray>
Regular Expressions	<regex>
Strings	<charconv> <format> <string> <string_view>
Thread Support	<barrier> <condition_variable> <future> <latch> <mutex> <semaphore> <shared_mutex> <stop_token> <thread>
Utilities	<any> <bitset> <chrono> <functional> <initializer_list> <optional> <tuple> <typeinfo> <type_traits> <utility> <variant> <memory> <memory_resource> <new> <scoped_allocator> <limits> <exception> <stdexcept> <system_error>

C++ Standard Library Headers

Zeiteinheiten <chrono>

- Unterschiedliche Zeitquellen
  - system_clock
  - steady_clock
  - high_resolution_clock
- Vordefinierte Zeiteinheiten
  - Clock::time_point
  - Clock::duration
- Beispiele

```
using Clock = chrono::system_clock;
Clock::time_point start = Clock::now();
Clock::duration d = Clock::now() - start;
int64_t ns = std::chrono::nanoseconds(d).count();
using ms_t = std::chrono::duration<double, std::milli>; // new duration type
double ms = std::chrono::duration_cast<ms_t>(d).count();
```

Container (1)

- Bitvektoren fixer Länge
  - bitset: <bitset>
- Halbdynamische Container
  - vector und vector<bool>: <vector>
- Listen
  - double ended queue: <deque>
  - list (doubly-linked): <list>
  - forward_list (singly-linked): <forward_list>
- Geordnete Mengen: <set>
  - set (die Schlüssel werden sortiert verwaltet)
  - multiset (Mehrfacheinträge sind erlaubt)
- Geordnete Maps: <map>
  - map
  - multimap (Schlüssel müssen nicht eindeutig sein)

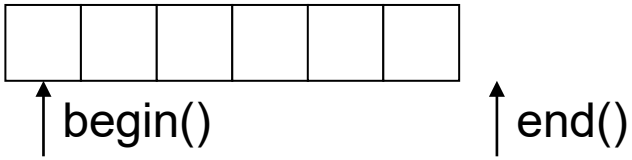
Container (2)

- Ungeordnete Mengen: <unordered_set>
  - unordered_set (die Schlüssel werden unsortiert verwaltet):
    unordered_multiset (Mehrfacheinträge sind erlaubt)
- Ungeordnete Maps: <unordered_map>
  - unordered_map
  - unordered_multimap (Schlüssel müssen nicht eindeutig sein)
- Container-Interfaces
  - verwendet einen Container (z.B., vector, deque oder list) als Datenbehälter
  - bietet spezielle Datenzugriffe an
  - Interfaces
    - stack (LIFO): <stack>
    - queue (FIFO): <queue>
    - priority queue: <queue>

Container: Datentypen und Methoden

- Datentypen (angeboten/erforderlich) für Container X<T>
  - X::value_type — Container-Element, entspricht T
  - X::reference — Referenz auf Container-Element
  - X::const_reference — dito, aber nur lesend verwendbar
  - X::iterator — Iterator
  - X::const_iterator — dito, aber nur lesend verwendbar
  - X::difference_type — vorzeichenbehafteter integraler Typ
  - X::size_type — vorzeichenloser integraler Typ für Grössenangaben
- Methoden (nicht vollständig)
  - Standard-, Kopier- und Verschiebekonstruktor, Destruktor
  - Iteratoren (lesend und schreibend): begin() und end()
  - Iteratoren (nur lesend): cbegin() und cend()
  - Grössenangaben: max_size(), size(), empty()
  - Zuweisungsoperator und Verschiebezuweisungsoperator
  - Relationale Operatoren
  - Vertauschen: swap(X&)

Iteratoren



- Konzept
  - Iterator: verallgemeinerter Zeiger, welcher auf ein Element des Containers zeigt
  - begin() und cbegin() liefern einen Zeiger, der aufs erste Element zeigt
  - end() und cend() liefern einen Zeiger, auf ein fiktives Element unmittelbar nachfolgend dem letzten Element
  - Inkrementieren ++ springt zum nächsten Element
  - Dereferenzieren * ermöglicht Zugriff aufs Element

- Beispiel

```
template<class Iter> void print(Iter it, Iter end) {
    while(it != end) {
       cout << *it++ << ' ';
    }
    cout << endl;
}
```

```
vector<int> v(10);
for(size_t i = 0; i < v.size(); i++) {
     v[i] =  i;
}
print(v.cbegin(), v.cend());
```

Iterator-Operationen

Operation	Input	Output	Forward	Bidirectional	Random Access
=	•		•	•	•
==	•		•	•	•
!=	•		•	•	•
*	1)	2)	•	•	•
->	•		•	•	•
++	•	•	•	•	•
--				•	•
[ ]					3)
arithmetisch					4)
relational					5)

1) Dereferenzierung ist nur lesend möglich.
2) Dereferenzierung ist nur auf der linken Seite einer Zuweisung möglich.
3) `I[n]` bedeutet `*(I+n)` für einen Iterator `I`
4) `+ += - -=` in Analogie zur Zeigerarithmetik
5) `< > <= >=` relationale Operatoren

Algorithmen

- Grundsätze
  - alle im Header <algorithm> vorhandenen Algorithmen sind unabhängig von einer konkreten Container-Implementierung
  - enthält eine Container-Implementierung einen gleichnamigen Algorithmus wie im Header <algorithm>, so soll die spezielle Version des Containers verwendet werden (höhere Effizienz)
  - die Algorithmen greifen über Iteratoren auf die Elemente des Containers zu
  - wird ein First- und ein End-Iterator verlangt, so ist damit das halboffene Intervall [First, End) gemeint
- Beispiel

```
const int searchValue = 5;
vector<int> v = { 9, 3, 5, 8, 1, 7, 2, 4 };

sort(v.begin(), v.end());
// get iterator to first element >= searchValue
auto pos = lower_bound(v.cbegin(), v.cend(), searchValue);
cout << *pos << endl;
```

Algorithmen: Übersicht (1)

- Suchen eines Elementes
  - find, find_if, find_end, find_first_of, adjacent_find
  - nth_element: platziert das n-te Element einer Sortierreihenfolge an die richtige Position im Array (z.B. um den Median zu bestimmen)
- Suchen einer Sequenz
  - search, search_n
- Zählen von Elementen, die ein Prädikat erfüllen
  - count
- Vergleichen zweier Elemente
  - min, max, min_element, max_element
- Vergleichen zweier Sequenzen
  - lexicographical_compare
- Vergleichen zweier Container
  - mismatch, equal

Algorithmen: Übersicht (2)

- Kopieren der Elemente eine Quellbereichs in einen Zielbereich
  - copy, copy_backward
- Vertauschen von Elementen oder Containern
  - swap, iter_swap, swap_ranges
- Einfüllen von Sequenzen
  - fill, fill_n, generate, generate_n
- Ersetzen von Elementen
  - replace, replace_if, replace_copy, replace_copy_if
- Entfernen
  - remove, remove_if, remove_copy, remove_copy_if
  - unique, unique_copy
- Transformieren (Kopieren und dabei Modifizieren)
  - transform

Algorithmen: Übersicht (3)

- Reihenfolge verändern
  - reverse, reverse_copy, rotate, rotate_copy, random_shuffle
  - partition, sort, partial_sort
- Permutationen
  - prev_permutation, next_permutation
- Suchen in sortierten Sequenzen
  - binary_search, lower_bound, upper_bound
  - equal_range
- Mischen zweier sortierter Sequenzen
  - merge, inplace_merge
- Mengenoperationen auf sortierten Strukturen
  - includes, set_union, set_intersection, set_difference, set_symmetric_difference
- Heap-Algorithmen
  - pop_heap, push_heap, make_heap, sort_heap

Exceptions (1)

- Werfen von Exceptions
  - Syntax: throw *ex-object;*
  - vordefinierte Exception-Typen in <exception>-Header
  - *ex-object* kann von jedem Typ sein, auch primitiver Datentyp
- Beispiel

```
try {
      throw std::runtime_error("example");
} catch(const std::runtime_error& e) {
      std::cout << "std::runtime_error: " << e.what() << std::endl;
} catch(…) {
      std::cout << "unknown exception" << std::endl;
}
```

Exceptions (2)

- Best-Practice
  - Exceptions nur by-value werfen (automatischer Speicher)
  - Exceptions als const-Referenzen auffangen
  - nur Exceptions abgeleitet von std::exception werfen
- Exception modifizieren und weiterwerfen
  - in catch-Block das Exception-Objekt modifizieren und mit throw weiterwerfen
- noexcept
  - eine Funktion kann deklarieren, dass sie niemals eine Exception werfen wird
  - dient der Performance-Optimierung
  - Move-Semantik benötigt noexcept (würde eine Move-Operation fehlschlagen, so wären sowohl das alte als auch das neue Objekt in einem invaliden Zustand)
- Konstruktoren/Destruktor
  - Konstruktoren können Fehlschlag nur über Exceptions kommunizieren
  - während des Exception-Handlings werden evtl. Destruktoren von Attributen aufgerufen
  - Destruktoren dürfen nie Exceptions werfen

Funktionale Elemente von C++

- Funktion
  - typisierte Parameterlisten
  - variable Anzahl Parameter
  - global oder als Methode einer (unveränderbaren) Klasse
- Funktor
  - Klasseninstanz, welche den Funktionsoperator operator()(…) überlädt
- Funktionszeiger
  - Adresse auf eine Funktion
- Methodenzeiger
  - Adresse auf eine an eine Instanz gebundene Methode
- Lambda
  - anonymer Funktor (kann auch innerhalb einer Funktion definiert sein)
- Funktionsobjekt
  - Verallgemeinerung all dieser Konzepte
  - Instanz der Klasse functional aus dem Header <functional>

Lambda

- Syntax
  - Zugriffsdeklaration Parameterliste [−> Rückgabetyp] Funktionskörper
- Beispiel
  - `[bias] (int x, int y) -> int { return bias + x + y; }`
- Zugriffsdeklaration
  - gibt in eckigen Klammern an, auf welche Variablen der Umgebung zugegriffen werden kann
- Parameterliste
  - Deklaration der Funktionsargumente analog zu normalen Funktionen
- Rückgabetyp
  - die Angabe des Rückgabetyps ist optional (kann vom Compiler selber ermittelt werden), darf auch void sein (Prozedur)
- Funktionskörper
  - ein gewöhnlicher Funktionskörper mit oder ohne return-Anweisung

Lambda Zugriffsdeklaration

- Hintergrund
  - dort wo der Lambda-Ausdruck definiert wird, existiert eine lokale Umgebung bestehend aus lokalen Variablen und Instanzvariablen
  - in der Zugriffsdeklaration wird angegeben, auf welche Variablen der Umgebung zugegriffen wird und ob der Zugriff by-value oder by-reference stattfinden soll
  - auf statische und globale Variablen kann immer zugegriffen werden, auch ohne Angabe in der Zugriffsdeklaration
- Beispiele
  - [bias] auf die Variable bias wird by-value zugegriffen
  - [&bias] auf die Variable bias wird by-reference zugegriffen
  - [=] auf alle Variablen der Umgebung wird by-value zugegriffen
  - [&] auf alle Variablen der Umgebung wird by-ref. zugegriffen
  - [this] auf alle Member der übergebenen Instanz wird by-pointer zug.
  - [=, &bias] nur auf bias wird by-ref. zugegriffen, sonst by-value
  - [factor, &bias] auf factor wird by-value und auf bias by-ref. zugegriffen

Funktionsobjekte im Einsatz

```
#include <functional> // … <vector>, <numeric>
void main() {
   // Deklaration des Funktionsobjekts
   function<float (float a, int x)> func;
   vector<int> v{1, 2, 3, 4, 5};

   func = … // Definition des Funktionsobjekts
            // (Funktor, Funktionszeiger, Methodenzeiger, Lambda)
            // siehe nächste Folie

   // Einsatz des Funktionsobjekts in einem Algorithmus
   float r = accumulate(v.cbegin(), v.cend(), 1.0f, func);
}
```

Verschiedene Funktionsobjekte

Funktor, Funktionszeiger

```
struct Funktor {
   float m_div;
   Funktor(float f) : m_div(f) {}
   float operator()(float a, int x) const
   {
      return a + x/m_div;
   }
};
----
func = Funktor(2.0f);
----
float foo(float a, int x) { return a +
x/2.0f;}
func = &foo;
```

Methodenzeiger, Lambda

```
struct C {
   float m_div;
   C(float f) : m_div(f) {}
   float meth(float a, int x) const {
         return a + x/m_div;
   }
};
----
C c(2.0f);
func = bind(&C::meth, &c, _1, _2);
----
func = [](float a, int x) { return a +
x/2.0f; };
```