Programmieren in C++

Strukturierte Datentypen und Klassen

Inhalt

- Erzeugen von Instanzen (Objekten)
- Lebensdauer von Instanzen
- C
  - Speicherbedarf von Instanzen
  - Bitfelder
- C++
  - Aufzählungsklassen
  - Klassen
  - Destruktor
  - Konstruktoren
  - Weitere Features
  - Variant (typsichere Form der Union)

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
```

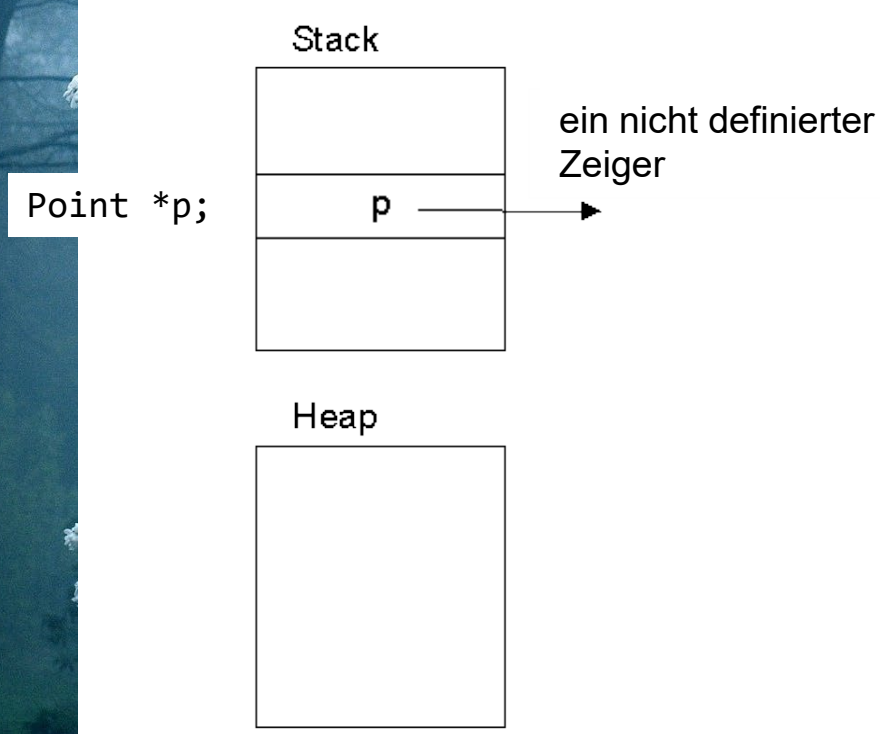
- Klasse

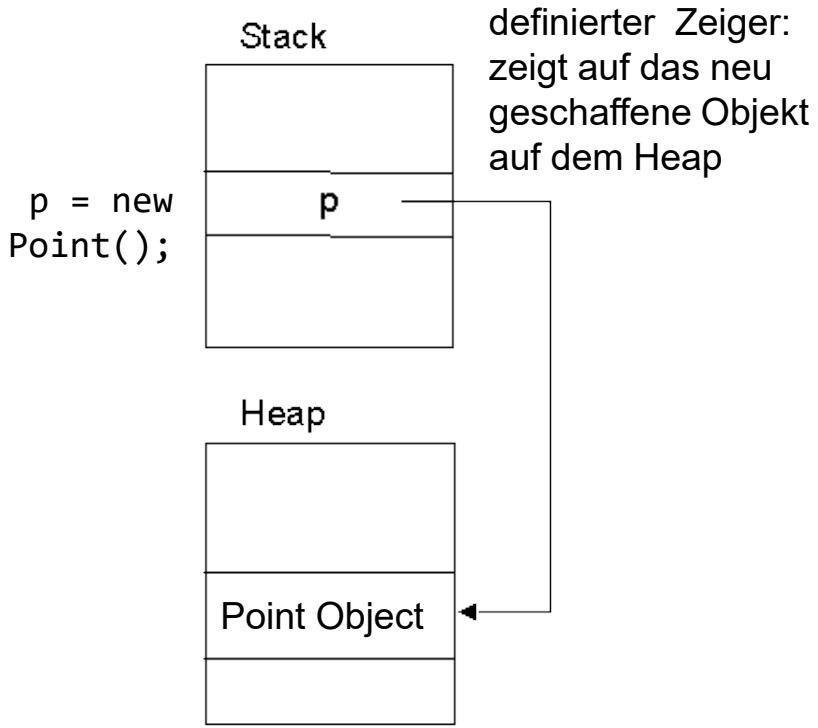
```
class Person {
  std::string m_name;     // private Attribute (Members, Instanzvariablen)
  int m_age;
public:             // ab hier wird die Sichtbarkeit auf public geändert
  Person(const char name[], int age); // Konstruktor
  std::string getName() const;
  void setAge(int age);
};
```

Erzeugen von Instanzen (Objekten)

- Objektgrösse
  - Compiler berechnet aus der Deklaration der Klasse, wie viel Speicher eine Instanz der Klasse benötigt
    - bei der Klasse Point sind zwei Attribute vom Typ int vorhanden
    - Konstruktoren und Methoden benötigen im Normalfall keinen Speicherplatz innerhalb von Instanzen
    - Speicherbedarf pro Punktinstanz: 2*sizeof(int)
- Objektallokation
  - im Speicher wird entsprechend Platz reserviert, so dass alle Attribute des Objektes abgespeichert werden können
- Objektinitialisierung
  - Konstruktor ist zuständig für die Initialisierung der Attribute des Objekts

Erzeugen einer dynamischen Instanz





Lebensdauer von Instanzen

- 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

Beispiel zur Lebensdauer von Objekten

```
static Point sp;  // Attribute werden automatisch mit 0 initialisiert

void main() {
   {
      Point p;     // Attributinitialisierung hängt vom Standard-ctor ab
      p.m_y = 10;
   }               // p wird hier automatisch zerstört
   Point* pp = nullptr;
   if (!pp) {
      pp = new Point(sp);  // erzeugt Kopie von sp auf dem Heap
      pp->m_x = 3;
   }
   pp->m_y = 5;
   delete pp;              // zerstört das Punktobjekt auf dem Heap
   if (Point* ip = new Point(); ip) {
      ip->m_x = 3;
      delete ip;
   }
}
```

Speicherbedarf von Instanzen

- Offset der Attribute in einem struct (relevant beim Speichern von binären Dateien)
  - die Reihenfolge der Attribute in der Deklaration ist massgebend
  - Offset eines primitiven Attributs ist ein Vielfaches seiner eigenen Grösse (= aligned)
- Speicherbedarf einer Instanz eines structs
  - Summe des Speicherbedarfs der einzelnen Attribute + Speicher für allfälliges Padding
  - gesamter Speicherbedarf ist ein Vielfaches des grössten primitiven Attributs
- Beispiel

```
struct S1 {
    int64_t d;
    int32_t i;
    int16_t s;
    int8_t c1;
    int8_t c2;
};
```

```
struct S2 {
    int8_t c1;
    int64_t d;
    int32_t i;
    int16_t s;
    int8_t c2;
};
```

Alignment und Packing

- Offset der Attribute in einem struct)
  - Alignment der Attribute kann durch Packing reduziert werden
- Speicherbedarf einer Instanz eines structs
  - natürliches Instanz-Alignment wird durch grösstes primitives Attribut bestimmt (kann durch Packing verringert werden und/oder alignas erhöht werden)
- Beispiel

```
#pragma pack(1)          // überschreibt Standard-Packing: 1 Byte Alignment verwenden
struct alignas(4) S {    // verwendet 4 Byte Alignment für Instanzen von S
    int8_t c1;
    int64_t d;
    int32_t i;
    int16_t s;
    int8_t c2;
};
#pragma pack()           // auf Standard-Packing zurücksetzen
static_assert(sizeof(S) == 16, "wrong packing"); // Zusicherung während des Kompilierens
static_assert(alignof(S) == 4, "wrong alignment"); // Zusicherung während des Kompilierens
```

Alignment der Attribute	1 Byte	2 Byte	4 Byte	8 Byte
Speicherbedarf von S	16 Byte	18 Byte	20 Byte	24 Byte

Bitfelder

- Syntax (innerhalb eines structs)

  Datentyp Name : *Konstante* ;

  - Einschränkungen
    - erlaubte Datentypen sind: char, int, long, Aufzählungstyp
    - Konstante darf nicht grösser sein als die Anzahl Bits des Datentyps
- Zugriff auf Felder
  - über Punkt-Operator
- Beispiel

```
struct {
    unsigned short character    : 8;
    unsigned short color        : 4;
    unsigned short underline    : 1;
    unsigned short blink        : 1;
} text[80];

text[20].character = 'a';
text[20].color = 5;
```

Inhalt

- C
  - Speicherbedarf von Instanzen
  - Bitfelder
- C++
  - Aufzählungsklassen
  - Klassen
  - Destruktor
  - Konstruktoren
  - Weitere Features
  - Variant (typsichere Form von Union)

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;
```

Deklaration der Klasse Point

```
// Datei: point.h
#pragma once
#include "color.h" // enthält Definition von Color

class Point {
    // private instance variables (attributes, members)
    double m_x, m_y, m_z;
    Color m_color;
public:
    // public instance methods
    Color getColor() const { return m_color; }   // inline-getter
    void print() const;
};
```

Implementierung der Klasse Point

```
// Datei: point.cpp
#include "point.h"
#include <iostream>

using namespace std;

// Implementierung der Methode print() aus der Klasse Point
// die Signatur muss der vorgegebenen in der Klasse Point entsprechen

void Point::print() const {
        cout  << "Punkt (" << m_x << "," << m_y << "," << m_z  << ") hat Farbe "
              << getColor() << endl;
}
```

Verwendung der Klasse Point

```
#include "point.h"

int main() {
    {
        Point p;       // statisches Punktobjekt auf dem Stack anlegen
        p.print();     // Punktobjekt auf Konsole ausgeben (Attribute sind nicht initialisiert)
    }                  // Punktobjekt p wird hier automatisch zerstört

    unique_ptr<Point> up;      // Zeigerobjekt
    {
        // dynamisches Punktobjekt auf dem Heap anlegen
        up = make_unique<Point>();
        up->print();           // Punktobjekt auf Konsole ausgeben
    }
    up->print();               // erlaubt
} // Punktobjekt auf dem Heap wird automatisch zerstört
```

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)
{ }
```

Verwendung von Konstruktoren

- Direkte Initialisierung

```
Point p1;                          // mit Standard-Konstruktor
Point p2(1, 2, 3);                 // mit Default-Farbe blau
Point p3(4, 5, 6, Color::green);   // mit gesetzter Farbe
```

- Kopie-Initialisierung wird zu direkter Initialisierung (copy elision)

```
Point p4 = Point(7, 8, 9, Color::red);   // weder Zuweisungsoperator noch
auto p5 = Point(1, 2, 3);                // Kopierkonstr. werden verwendet
```

- Punktobjekte auf dem Heap allozieren

```
Point *pp1 = new Point(1, 2, 3);
auto *pp2 = new Point(4, 5, 6, Color::yellow);
auto up = make_unique<Point>(7, 8, 9, Color::white);
auto sp = make_shared<Point>(1.0, 2.1, 3.2);
```

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
```

Anonyme (temporäre) Objekte

- Objekte haben keinen Namen
- werden nur kurzfristig (temporär) benutzt
- nach dem Aufruf verschwinden statische, anonyme Objekte wieder
  → kurze Lebensdauer

Beispiele

```
Point(1, 2, 3, 4).print();          // anonymes Objekt wird nach dem Aufruf von
                                    // print() gleich wieder zerstört

(new Point(2, 3, 4, 5))->print();   // schlecht: verwaistes, anonymes dynamisches
                                    // Objekt lebt auf dem Heap weiter → Memory Leak
```

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

Beispiel zu Klassenvariablen

h-File

```
class Point {
    // instance variables
    double m_x, m_y, m_z;

    // class variable declaration
    static int s_nrOfInstances;
    static const int s_version = 111;

public:
    // ctor
     Point();

    // dtor
    ~Point();

    // class method
    static int getNrOfInstances();
};
```

cpp-File

```
// class variable definition
int Point::s_nrOfInstances = 0;

// ctor
Point::Point() : m_x(0), m_y(0), m_z(0) {
      s_nrOfInstances++;
}

// dtor
Point::~Point() {
     s_nrOfInstances--;
}

// methods
int Point::getNrOfInstances() {
     return s_nrOfInstances;
}
```

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
```

Typkonvertierungs-Konstruktor

- enthält üblicherweise nur ein Argument (wird mit nur einem Argument aufgerufen)
- kann zur impliziten Konvertierung verwendet werden
- soll ein Konstruktor mit nur einem Argument nicht zur impliziten Konvertierung missbraucht werden, so muss vor dem Konstruktor das Schlüsselwort explicit geschrieben werden

Beispiel

```
explicit Point(double p[3], Color c = Color::black)
: m_x(p[0]), m_y(p[1]), m_z(p[2]), m_color(c)
{ }
```

Anwendung

```
double array[3] = { 4.4, 3.3, 2.2 };
Point p(array);          // erlaubt, weil der Konstruktor nicht explicit ist
p = array;               // nicht erlaubt, weil expl. Konvertierung gefordert ist
```

Resource Acquisition is Initialization

- RAII
  - bindet den Lebenszyklus einer erworbenen Ressource an die Lebensdauer eines Objekts
- 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; }
};
```

RAII: Einfacher Lösungsansatz

- Idee
  - wird ein Objekt infolge einer Exception nicht vollständig initialisiert, so müssen die einzelnen Teile des Objektes sich selbständig abbauen
- Beispiel

```
struct StereoImage {
    std::unique_ptr<Image> left, right;
    StereoImage() : left(new Image), right(new Image) {}
};
```

- Was passiert bei einer Exception in right(new Image)?
  - StereoImage gilt als nicht erzeugt, daher wird der Destruktor nicht aufgerufen
  - bereits angelegter Speicher (left) wird zurückgebaut, indem allfällige Destruktoren der bereits angelegten Attribute aufgerufen werden
  - Speicher für das linke Bild wird freigegeben

Default-Methoden

- 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&)`

Weitere Features (1)

- Einheitliche Initialisierungssyntax
  - Point p{1, 2, 3}; // falls Punkt keinen eigenen Konstruktor besitzt,
    // so wird Aggregats-Initialisierung verwendet;
    // das kann verhindert werden, indem der Konstruktor
    // mit runden Klammern direkt aufgerufen wird
  - im Konstruktor: Point::Point() : m_x{0}, m_y{0}, m_z{0} {}
- Instanzvariablen-Initialisierung (wie in Java)
  - class Point {
    double m_x = 1, m_y(1), m_z{1}; // alle 3 Varianten möglich
- Konstruktor-Delegation
  - besitzt eine Klasse mehrere Konstruktoren, so darf der eine Konstruktor den anderen in seiner Initialisierungsliste aufrufen

    Point::Point() : Point(0, 0, 0, Color::white) { }

Weitere Features (2)

- Konstruktor/Zuweisungsoperator trotzdem generieren
  - wenn Sie mit dem, was der Compiler generieren würde, zufrieden sind, dann können Sie ihm mit = default sagen, dass er automatisch einen Konstruktor/Zuweisungsoperator generieren soll

```
class C {
    C(int x) { }        // benutzerdefinierter Konstruktor unterdrückt
                        // automatisch generierten Standard-Konstruktor
    C() = default;      // generiert trotzdem Standardkonstruktor
};
```

- Automatisch generierte Konstruktoren/Zuweisungsoperatoren löschen
  - möchte man beispielsweise die Verwendung des Kopierkonstruktors und des Zuweisungsoperators verunmöglichen, weil nur die Move-Semantik verwendet werden soll, so kann dem Compiler die Löschung dieser automatisch generierten Methoden beauftragt werden

```
C(const C&) = delete;
```

Variant (typsichere Form der Union)

- Idee
  - ein Objekt der Klasse Variant speichert ein Objekt, dessen Typ aus einer Menge von zulässigen Typen stammt
- Syntax
  - std::variant<T1, T2, …, Tn> var; // default-ctor von T1 wird verwendet
- Zugriff auf gespeichertes Objekt
  - std::get<T>(var) // wobei T aus T1 bis Tn stammen muss
- Abfrage der aktuellen Variante
  - var.index() // gibt 0-indizierten Index der Variante zurück
- Beispiel

```
std::variant<int, float> vif(5.5f);
std::cout << std::get<float>(vif) << std::endl;
std::cout << "index = " << vif.index() << std::endl;
std::cout << std::get<int>(vif) << std::endl; // wirft bad_variant_access ex.
vif = 4;
std::cout << std::get<int>(vif) << std::endl;
std::cout << "index = " << vif.index() << std::endl;
```