**Von C zu C++** (g++ –std=c++14 –Wall –pedantic-errors)

#include <string> (C++-Konvention: Angabe ohne .h)

**Datentyp string**

#include <string>

using namespace std;

string s; string t = "abc"; t[1] = 'x';

s = t + "def"; s += "xyz"; cout << "s = " << s;

**Dynamische Speicherreservierung**

* double\* f2 {new double {3.14}}; delete f1;
* int\* numbers {new int[42]}; delete[] numbers;

**Verschiedene Änderungen und Erweiterungen**

* **nullptr** statt **NULL**
* Datentyp **bool**
* **range-based loop** **for (auto e: collection)** process(e);
* **Scope-Operator**:: (auf globale VAR durch den Scope-O zugreifen) Bsp: ::x = 17;
* **Überladen von Funktionen** ist möglich!

**Parameter mit Defaultwert**

* void saveToFile(string path, **bool ovIfExists = false, string hd = "File:** ");

**Streams** #include <iostream>

* Ausgabe Operator **<<** || **std::cout** vom Typ istream
* Eingabe Operator **>>** || **std::cin** vom Typ ostream
* Syntax: **cout <<** **Ausdruck** bzw. **std::cin >> Variable**

Standardmäßig keine Exceptions, sondern Fehlerbit in Stream abfragen und anschließend zurücksetzen:

while (!cin.fail()) { cin >> n; ... } cin.clear();

**Exceptions**

* **throw exception("Fehler");** // new unüblich, da dann jemand delete'n müsste

**Namensraum std -** using namespace std;(am anfang)

* Nicht in Header-Dateien
* Ganz ohne using: Immer std::cout, std::endl etc. benutzen

**Referenz-Typen**

* Eine Referenz ist keine eigenständige Variable, sondern ein anderer Name ("Alias") für eine bestehende ("referenzierte") Variable (refferenz auf einen Zeiger)

int x;

int& a {x}; //a ist Alias für x

int& f(int& x) { x++; return x;}

for ( **const** auto& elem: collection) // keine Konstruktor-Aufrufe 

process(elem);

**Klassen in C++**

|  |  |
| --- | --- |
| #include <string>  #ifndef Test\_H  #define Test\_H  class Test {  public:  Test(const std::string &name);  Test(const Test& old);  const std::string &getName();  void toString()const;  private:  std::string name;  //static attribut  static int lastid;  const int itemID;  };  #endif | #include <string>  #include <iomanip>  #include <sstream>  #include "Test.h"  //Initialisierungsliste  Test::Test(const string &name) :itemID{++lastid}, name{name}{}  Test::Test(const Test& old):itemID{++lastid}{ //weil const!  this->name = old.name; //copy-Konstruktor(tief)  }  const std::string &Test::getName() {return this->name;}  void Test::setName(const std::string &name) {  this->name = name;  }  std::string CartItem::toString() const{  std::ostringstream stream;  stream << " x ";  return stream.str();  }  //static attribut  int Test::lastid {0}; |

**Destruktor-Methode Copy-Konstruktor**

* Aufgabe des Destruktors: Freigabe von Ressourcen

|  |  |
| --- | --- |
| Person::**~**Person(){  **delete(this->name);**  **this->name = nullptr;**  } | Ort::Ort(**const Ort& old){**  this->breite = old.breite;  initName(old.name);  } |

**const-Methoden (Beobachtermethoden) Klassenatribute static**

* Eine Methode mit const darf kein Attribut des Objekts verändern.

|  |  |
| --- | --- |
| public:  void print()**const**;  void Person::print()**const**{  std::cout << this->name << std::endl;  } | private:  **static int last\_id;**  **const int id;** ...  **int Ort::last\_id = 0;**  Ort::Ort(…):**Id{last\_id++}** { ... |

**toString-Methode**

|  |  |
| --- | --- |
| #include <string>  class Person {  public:  **std::string toString()const;** | #include <string>  #include <sstream>  std::string Person::toString() **const**{  **std::ostringstream os{};**  **os** << "Name :"<< this->name;  return **os.str();**  } |

**Copy-Konstruktor, Destruktor und Zuweisung**

* "Dreier-Regel": Klassen benötigen entweder keine oder alle drei dieser Methoden

**Operatoren**

Syntax: **Methodenname operator op(...) #include <iostream>**

Beispiele:

|  |  |
| --- | --- |
| public: ...  **Vektor operator + (const Vektor& v) const;**  double operator \* (const Vektor& v) const;  Vektor &operator = (const Vektor &v);  Vektor &operator += (const Vektor &v);  bool operator < (const Vektor &v); | #include "vektor.h"  **Vektor Vektor::operator + (const Vektor& v){**  **Vektor res {};**  for (int i = 0; i < N; ++i)  res.elems[i] = this->el[i] + v.el[i];  return **res**;  } |

**friend-Deklarationen**

* Ausweg: Klassenfremden Methoden privaten Zugriff einräumen mit friend:

|  |
| --- |
| **class Vektor {**  **friend ostream& operator << (ostream& out, const Vektor& v);**  **private:**  **};** |

**Ausgabe-Operator << (**als globale Funktion mit Parametern (ostream, Vektor))

|  |
| --- |
| **//main.cpp**  **ostream& operator << (ostream& stream, const Vektor& v) { //in \*.h deklarieren**  **stream << "(" << v.elems[0] << ", " << v.elems[1] << ")";**  **return stream;**  **}** |

**Eingabe-Operator >>**

|  |  |
| --- | --- |
| **istream& operator >> (istream& in, Vektor& v) {**  **in >> v.elem[0] >> v.elem[1]**  **return in;**  **}** | **int main(void) {**  **Vektor v {...};**  **cout << "v = " << v << endl;** |

**Index-Operator**

* Erlaubt Benutzung wie ein eingebauter Array

|  |  |
| --- | --- |
| **double& Vektor::operator [] (int index) {**  **if (index < 0 || index >= N)**  **throw invalid\_argument("Indexfehler");**  **return elems[index];**  **}** | **//main.cpp**  **Vektor v = ..., w = ...;**  **v[1] = 3.14; double x = (v+w)[2];**  **cout << v[0] << endl;** |

**Vererbung**

class CPos : **public Pos** { public: ...

**Polymorphie (Dynamisches und statisches Binden)**

* Voraussetzungen: Die Methode wird über einen Zeiger oder eine Referenz auf das Objekt aufgerufen

🡪 **Dynamisches Binden mit virtual**

|  |  |  |
| --- | --- | --- |
| // Grobject.h  class Grobject {  private: CPos cpos;  public:  Grobject(int c, int r,  Color col);  virtual ~Grobject();  **virtual draw() const override;**  }; | Class Hline: **public Grobject** {  public:  Hline(int c, int r, Color col, int l);  virtual ~Hline();  **virtual draw() const override;**  private:  int len;  ; | Hline::Hline(...):  **Grobejt{c,r,col},** len{l){}  void Hline::draw() const {  **Grobject::draw();**  for( ... ){  std::cout << “-"; |
| // in main.cpp  Rect a {10, 5, BLUE, 20, 4};  Grobject\* ap = &a;  ap->draw(); | // in main.cpp  Rect b {20, 7, YELLOW, 5, 10};  Grobject& br = b;  br.draw(); |

**Abstrakte Klassen und Methoden**

* In C++ durch Methodenkopf = 0;

|  |  |
| --- | --- |
| // Grobject.h  class Grobject {  private: CPos cpos;  public:  Grobject(int c, int r,Color col);  virtual ~Grobject();  **virtual draw const = 0;**  }; | //Grobject.cpp  ...void Grobject::draw() const { prep(); }; |
| void Hline::draw() const {  **Grobject::draw();**  for( ... ){  std::cout << “-"; |

**Downcasting dynamic\_cast**

|  |  |
| --- | --- |
| Rect r {10, 5, BLUE, 20, 4};  Grobject\* gp = &r; ...  Rect\* rp = (Rect\*)gp // Downcast  cout << "Breite = " << rp->w; | **Rect r {10, 5, BLUE, 20, 4};**  **Grobject\* gp = &r; ...**  **Rect\* rp = dynamic\_cast<Rect\*>(gp); // Downcast if (rp != nullptr) cout << "Breite = " << rp->w;** |
|

**Mehrfachvererbung**

Syntax : class TRect : **public Rect, public Text** { …

🡪 Mehrdeutigkeit!! Mehrfachvererbung ist grundsätzlich kompliziert und fehlerträchtig

* **Statisches Binden**: Methodenauswahl je nach Variablentyp
* **dynamische Auswahl** nach tatsächlichem Objekttyp

|  |  |
| --- | --- |
| **Java**   * Alle Methoden werden zunächst dynamisch gebunden! * **final** 🡪 statisch gebundenen Methoden * In Java gibt es nur Referenzvariablen | **C++**   * Alle Methoden werden zunächst statisch gebunden * Methoden können durch das Schlüsselwort **virtual** zu dynamisch gebundenen Methoden * über Zeiger- und Referenzvariablen! |