

operator +: (homogenes Beispiel)

```

Date:: Date operator + (int days) {
    Date dt;
    dt._day = day + days;
    dt._month = month;
    dt._year = year;
    if (dt._day > 30) {
        dt._month += dt._day / 30;
        dt._day = dt._day % 30;
    } if (dt._month > 12) {
        dt._year += dt._month / 12;
        dt._month = dt._month % 12;
    }
    return dt;
}

```

abstrakte Methode:

- virtuelle Methode, ohne Implementierung in der Basisklasse → rein virtuelle Methode
- virtual void methode () = 0;
- zuech: erkennt, dass abgeleitete Klassen eigene Version der Methode implementieren müssen

abstrakte Klasse:

- abstrakte Klasse enthält mindestens eine rein virtuelle Methode
- können teilweise implementierte sowie vollständig virtuelle Methoden enthalten, nur min 1 rein virtuelle
- Zweck: Basis für andere Klassen, definieren gemeinsame Schnittstelle
- Hinweis: Es können nicht direkt Objekte einer abstrakten Klasse erzeugt werden, da Objekt unvollständig wäre, es hätte unvollständige Methoden

Interface:

- rein abstrakte Klasse, alle Methoden rein virtuell, enthalten keine Member-Variablen
- von rein abstrakten Klassen können keine Instanzen erstellt werden
- abgeleitete Klassen müssen alle rein virtuellen Methoden der Basisklasse implementieren um überarbeitet werden zu können, erst dann sind abgeleitete nicht mehr abstrakt
- Zweck: Schnittstelle für andere Klassen, indem sie ausschließlich Methoden deklarieren

ostream operator < : 2. Vr.

metello
Wörter
-> friend ostream & operator <<(ostream &, const Date &);
ausdrücklich
Wörter
-> ostream & operator <<(ostream &, const Date&, const Date&, date);
return cout << date._day << ":" << date._month <<
" " << date._year; ;

da friend
von Klasse num
wird extra Date:
angewiesen
werden!

Algorithmus Dynamic_cast:

- a) Es wird geprüft, ob Element auf Objekt Lebensmittel zeigt

- b) wenn ja Daten des Datums < heute,
wenn größer entfernen am Lager
b) wenn vorneheute (hier Lebensmittel)
nichts zu tun

-> void entnahmefen(Warenlager & lager);

Ware* Ware;
for (int i = 0; i < lager.bdegkepliste(); i++) {
Ware = lager[i];

Lebensmittel * L = dynamic_cast<Lebensmittel>(Ware);
if (L != nullptr) {
 Pointe auf Lebensmittel zeigt
 Datum mhd = L->get_mhd(); // Datum zugewiesen
 if (mhd < Datum::heute()) {
 lager.entnahme(L);
 verglichen
 }
}

Static_cast:

- static_cast = Typ den man will > (Vor. die. umgewandelt)
→ z.B. int x in double umwandeln, ist y nicht.
→ double = static_cast<double>(x) / y;
x explizit → y wird implizit umgewandelt

dynamic_cast:

- benötigt mindestens eine virtuelle Methode für Polymorphie
→ wandelt vor allem Pointe um
→ Base * b = &Base; i in ^{wandler b aus Base} Derived * zeigend
→ Derived * d = dynamic_cast<Derived>(b);

vererbliche Liste:

→ struct ListElement {

int id;
ListElement * p-next;
};

ListElement * p-root = nullptr;

→ void add_new_head_element (int id) {

ListElement * p-new-element = new ListElement();
p-new-element → id = id;
p-new-element → p-next = p-root;
p-root = p-new-element; ;

std::exception ala ein, überstricken what-int.

- abdrücken von exception mit Myexception und Konstruktor mit Par. über.

→ überschriften what() met. bereitst. fehlermeldung

→ #include <exception>

class Myexception : public exception {

String datei, Beschreibung;

int zeile;

Myexception (String dt, int z, String b)

: datei(dt), zeile(z), beschreibung(b) {}

→ const char* what() const noexcept override {

return beschreibung; C-Str(); } ;

altru: return ("Fehler Datei" + datei + ", Zeile"

+ to_string(zeile) + " Beschreibung " + beschreibung)

.C-Str(); ← muss nicht so gemacht werden

einfach anders durch what ausgabe immer

mit .C-Str(); beenden!!

→ nochmal wichtig Elefantenloho abgleichen

→ class Elefantenloho : public my_exception {

public: Elefantenloho (String s) : Myexception(s, -1) {}

→ so wird gemacht:

→ throw MyException (- File -- , Line -- ,

" Fehler-Zooname ");

→ throw Elefantenloho (- File -- , Line -- ,

" Elefant auf Mars ");

→ void delete_element (int id) {

ListElement * p-temp = p-root;

ListElement * p-predecessor = nullptr;

while (p-temp != nullptr &&

p-temp->id != id) {

p-predecessor = p-temp;

p-temp = p-temp->p-next; ;

if (p-temp != nullptr) {

if (p-temp == p-root) {

p-root = p-temp → p.next; ;

else {

p-predecessor->p.next = p-temp->p.next; ;

delete p-temp; ;

void print_list () {

ListElement * p-temp = p-root;

while (p-temp != nullptr) {

cout << p-temp->id << endl;

p-temp = p-temp->p-next; ;

Template insertion cont: template (typename T)

void insertsoft (T arr[], int n) {

T key; int j;

for (int i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j+1] = arr[j];

j = j - 1; ;

arr[j+1] = key; ;

Linearer Suchalgorithmus: Template < typename T >

int linearsearch(T arr[], int n, T x) {

for (int i = 0; i < n; i++) { if (arr[i] == x) return i; return -1; }

Template Search: Template < typename T >

class Stack {

vector < T > elems;

public:

void push (T const& elem) { elems.push_back (elem); }

void pop () { if (!elems.empty ()) elems.pop_back (); }

T pop () const { return elems.empty () ? -1 : elems.back (); }

bool empty () const { return elems.empty (); }

Linked List: Template < typename T >

class Node {

public:

T value;

Node * next;

Node (T val) : value(val), next(nullptr) {}

→ class Linkedlist {

Node < T > * head;

public:

Linkedlist () : head(nullptr) {};

void add_new_head_element (int id) {

ListElement * p-new-element = new ListElement();

p-new-element → id = id;

p-new-element → p-next = nullptr;

if (p-root == nullptr) {

p-new-element = p-root; ;

else {

listElement * p-temp = p-root;

while (p-temp != nullptr) {

p-temp = p-temp->p-next; ;

listElement * p-new-element = new listElement();

p-new-element → id = id;

p-new-element → p-next = p-temp; ;

listElement * p-temp = p-temp->p-next; ;

Fragen und Antworten:

- Attribut static, const muss über Initialisierungshilfe initialisiert werden
X auf Konstruktor der Klasse, static Attribute, Klasseattribute aufgerufen
- Initialisierungshilfe kann im Punkt angeführt werden
X const darf nicht zugewiesen werden, nur in Initialisierungshilfe
- Für Tiefe Kopie, selbst definiert Kopierkonstruktor initialisiert
✓ andernfalls nur das erste Element, z.B. Anhänger einer verlinkten Liste
- Freund von Klasse darf auf protected zugreifen List Kopiert
✓ Freund einer Klasse darf auf alles zugreifen
- Konstruktor abg. Klasse als erster Defaultparameter. Basishl. aufgerufen
nen kann explizit anderen Konstr. der Basisklasse aufrufen
- operator= nicht als const-hilf.
Bei Konst Met. nicht erlaubt, da const nicht zugewiesen werden dürfen
- In map kann effizienter gearbeitet werden, als in Vektor
✓ Map organisiert sich selbst. Suche ist in $O(\log N)$ möglich
- a. Klasse B abg. Klasse A, Klasse C abg. Klasse B. Zeiger auf C (C^* ptr)
kein direkter Zugriff auf Obj. Klasse A zeigen (A^* a; C^* ptr = &a;)
X Obj. a nicht alle Eigenschaften, die Comp erwartet würde beim
Zugriff über ptr. → unerwartetes Verhalten
- b. Klasse mit virtuellen Methoden abstrakte Klasse
X nun mindestens eine rein virtuelle Methode enthalten
- Handler für Exception von Basisklasse hinter Handler-Exception von abgeleiteten
✓ System überprüft catch Blöcke der Basis nach, wenn Basisklasse darunter
wird diese nicht abfangen, spezifische Handler für Basisklasse würde nie
erreicht werden, wenn hinter Basisklasse. (Hinweis: andernfalls nach
Substitutionsprinzip auch Exception der abgeleiteten Klasse gefangen)
- Der Zugriff auf Zeichen eines Strings mit Methode at ist nicht
so effizient wie mit Operator[], dafür aber sicher.
✓ Die Methode at führt eine Bereichsprüfung durch und wirft ggf. eine exception
- static-Methoden der Klasse können nur aufgerufen werden, wenn es Objekt der Klasse gibt
X statische Methoden benötigen keinen This-Zeiger, auch aufrufbar, wenn kein Objekt gibt
- Aggregation, wenn aggregierende Objekt, Lebensdauer der aggr. Teilobj. kontrolliert
X In diesem Fall Komposition. Bei Aggregation Lebensdauer der Teilobjekte unabhängig
von der Lebensdauer des Gesamtobjekts
- Mit class A { int a; friend B; }; kann Klasse A auf private Elemente Klasse B zugreifen
X Klasse kann nur Freunde definieren, die Zugriff auf sie hat. Klasse kann nicht definieren
dass sie selbst Freunde von anderer Klasse (B) ist. In dieser Definition darf
B auf A zugreifen, aber nicht A auf B.
- In abgeleiteter Klasse kann neue Methode Basisklasse überladen X Funktion nur Name unterscheidet
im gleichen
- Kopierkonstruktor, kann Objekt call by Value-/Reference übergeben werden → nur call by Reference möglich
X Wenn Kopierkonstruktor Objekt call-by-Value übergeben, so müsste Inhalt in neue lokale
Variablen kopiert werden. Hierzu wiederum Konstruktorstr. nötig. Es liegen mehrfache Referenzen

Chalget operatoren bsp:

- operator wie Funk. anzusehen mit Typ return wert, logik überg. Wdh
- operator+=: Fügt Punkt hinzu vor Obj.
- Point operator+(const Point& other){
return Point(x+other.x, y+other.y);}
- operator==: vergleicht Punkte mit
{ zugehörigen Obj.}
- bool operator==(const Point& other){
return (x == other.x && y == other.y);}
- operator=:
operator =: neuen Wert zwisch. Obj. wird.
Point& operator=(const Point& other){
if (this != &other) {
x = other.x; y = other.y; }
return *this; }

Polymerphie:

- funktion zusammenhängen mit Polymerphie:
- Bierlaufs bl(a,b,c) → klassisch hl(ab,c);
bl.add(0,1,tr); hl.add(0,1,tr);
bl.add(1,2,tr); hl.add(1,2,tr);
bl.list(); hl.list();
bl ~ Bierlaufs(); hl ~ klassisch
- zusammenfassung:
- void test (Shatpiel & shat) {
shat.add(0,1,tr);
shat.add(1,2,tr);
shat.list();
shat ~ Shatpiel();}
- implementiert:
- test(bl); und test(hl);

16. Module werden separat kompiliert, nutzbar für module. export inner über import

17. Für Tiefe Kopie schwachdefinitor (Kopierkonstr.). Ju, ansons. 1. Eltern z.B. Anhänger List. kopiert

18. Zuweisungsoperator =(), nur als Met. von Klasse mit Globaler Funkt. oder Freunde-Funkt. auch möglich

19. Mehrstufige Vererbung verhindert durch virt. Met. Virtuelle Vererbung verwendet, nicht Methoden

20. Aggregation objekt class TestKfz { Kfz fahrt; };

Aggregation: Einzelns., Blatt unabhängig, Composition:
Ein Blatt Teil ganzen Baumes, teil un trennbares Baumelement

→ neuer Attribut in Klasse ist deren gebender

→ multiberes. gärtner, Composition, das

21. f() in B verdeckt f() in A verdecken bedeutet andere Met. kann nicht mehr aufgerufen, abgeleitete Klasse gleicher Funktionsnamen != überladen!

22. überschreiben virt. Met. Rücktypolog. dgl. gleich Rücktypen müssen kompatibel sein nicht exakt gleich, z.B. ohne Methode unterschl. Pointer

23. Exception nicht geklärt und Block sonst vorlassen, mit neu erzeugt. Obj. nicht geklärt im Obj. Heap Gr. der Obj. im try Block, für dinge die im Heap liegen. Konstruktor notwendig mit Dektor → Dektor-Stack implizit geklärt, beim Decap explizit

24. Initialisierung Konstruktor. er. Comp. Konstruktorcode Ja wegen Temp. Initialisierung 25. Ergebnis mit Bas. Klasse über Pointer addenziert Ja

Templats:

- Klassentemplate Kinder, in jedem Kindehr Daten, jeder Kindehr beliebig viele Kinder haben, Klassentemplate Wichter um alle Kinder von Kindehr zu spezifizieren
 - template < Typename T > (Orange as main)
 - klassen Kinder &
 - T data; vector < Kinder > Kinder;
 - public:
 - Kinder (T data): data(data) {}
 - Wurzel < int > h2(2); / Wurzel < string > c1("Hello");
 - getdata () & return data; h2.getdata();
 - int aus2(Wurzel()) & return Kinder.size(); h2.anzahl();
 - void platzherr (Kinder &h) & h1.platzherr (h2);
Kinder < int > -beruh (h);
 - void print (bool nl=true) & h1.print() / c1.print();
cout << data << " - ";
for (unsigned int i = 0; i < Kinder.size(); i++) &
Kinder[i].print(nl); if (nl) cout << endl; }
- Template Bubble Sort:
- ```
const unsigned unsigned int laenge = 0; // wenn Array und
template < Typename T > (T arr[], int n) // Länge n über,
array < T, laenge > sortiere (array < T, laenge > -feld) &
bool sortiert = false;
while (!sortiert) {
 sortiert = true;
 for (int i = 0; i < n-1; i++) {
 if (arr[i] > arr[i+1]) {
 T temp = arr[i];
 arr[i] = arr[i+1];
 arr[i+1] = temp;
 sortiert = false;
 }
 }
}
return -feld; }
```
- allg. Kontakt
- sortiert = true; (int i = 0; i < n-1; i++) {  
for (size\_t i = 0; i < -feld.size() - 1; i++)  
if (-feld[i] > -feld[i+1]) {  
T temp = -feld[i];  
-feld[i] = -feld[i+1];  
-feld[i+1] = temp;  
sortiert = false; } } return -feld; }

## Template Fibonacci:

- template < Typename T >
- T fibonacci (int n) & if (n <= 1) return ;  
return fibonacci < T > (n-1) + fibonacci < T > (n-2);

## Max, Min Funktion:

- template < Typename T >
- T max ( Ta, Tb ) & return (a>b) ? a:b; ;  
T min ( Ta, Tb ) & return (a<b) ? a:b; ;
- Sweaps: Template < Typename T >
- void swap ( T & a, T & b ) &  
T temp = a; a = b; b = temp; ;



## Konstruktor, Destruktor, Exceptions:

- Objekte bleiben seligen erhalten, wie die Funktion besteht in der sie erstellt werden
- Wenn der destruktur diese Lektur noch sie der Reihe nach wieder gelässt
- wenn Funktion durch throw erworben wird, werden zuerst Objekte gelöscht bevor exception geworfen
- int main() {
 try {
 Base base(5);
 func(1);
 catch (int e) {
 cout << "caught"; }
 }
 }

```
class Base {
 public: Konstruktor: create
 ~Base Destruktor: destroy();
 class Derived: public Base {
 Derived (int -id): Base (-id) { create(); }
 ~Derived () { destroy(); }
}
```

```
void func (int n) {
 cout << n;
 Derived derived (n);
 if (n == 0) throw 100;
 func (n - 1);
 cout << leave func(); }
```

1. create Base(), 2. enter func(), 3. create Base  
 4. create derived(), 5. enter func(), 6. create Base  
 7. create derived(), 8. destroy derived(), 9. return  
 Base(), 10. destroy derived(), 11. destroy Base(),  
 12. return 100, 13. destroy Base()

## Aufgabe 3: Vererbung:

- Konstruktor - Fall 1: Konstruktor mit Name und Datum als Parameter  
 → Lebensmittel:: Lebensmittel (string id, Datum mhd) : Ware (id), mhd (-mhd);
- Konstruktor - Fall 2: zusätzlich zu Name ein Prädikat: auch noch MHD hinzugefügt  
 → 3 Parameter entsprechend, delegieren an Basisklasse keine Kopien der Attribute vorgenommen bei double kontrahiert
- Fussweg (string& s, string& z, double e) : Strecke (s, z, e); § 3

überdecken Methode: legezurück soll überdecken  
 → void legezurück() const override;  
 → void Fussweg:: legezurück() const {
 cout << "lange Strecke < start << endl; }

## Überschreibbarer toString:

- string toString() const override; überdecken ohne
- string Lebensmittel:: toString() const {
 string result = Ware:: toString();
 result += ", MHD: " + const\_cast< Datum >(&mhd).toString();
 return result; }

## Spielstand angeben Überprüfung ob jemals -301:

- virt. Met. listSpielstand aufrufen, prüfen ob mindestens 1 Spieler mehr 301 mindestens, dann beenden
- void Bierdeckel:: listSpielstand() const {
 Shatzspiel:: listSpielstand(); → an alter Bas. Kl. aufrufen
for (Spieler \*s: Spieler) { → in Schl. alle checken
if (s-> getPunkte() < -301) { → get aufrufen
cout << "beendet" << endl;
break; } } }

## Zugriff auf Variablen:

- nur abgleichbare Werte z.B. zugriff auf Attribut string name möchte ist dies nicht grundsätzlich möglich nur mit public/protected, enservaten getter-Methoden aufrufen

## Vektor:

- erneuert auf Basis der Zeile
- clear() → löscht alle
- empty() → abgleicht ob Vektor leer ist
- resize (int) → ändert Größe Vektor auf n Elemente
- for (auto it = v.begin(); it != v.end(); ++it) {
 cout << \*it << " "; } → Angabe Vektor umgedreht Reihenfolge

## Aufgabe 6: Klassen, Operatoren, Vektoren:

- methode welche Pointe auf Ware Vektor hinweist:  
 → Ware Lager:: Lager (Ware\* w) & gibt Referenz zurück  
 → Ware Lager & Ware Lager:: Lager (Ware\* ware) {
 if (lager.size() < kapp) { → size() Länge des Vektors
 lager.push\_back (ware); }
 else { cout << "Lager voll" << endl;
 return \*this; }

## bedarf angeben Vektor:

- void Ware Lager:: listBedarf() const {
 cout << "Lager mit" << kapp << "Plätzen" << endl;
 for (int i = 0; i < lager.size(); i++) {
 for (auto & ware: lager) { → einfacher, list lager, r: vektor
cout << lager[i] >> toString() << endl; }
 cout << ware >> toString() << endl; }
- Bei bedarfsbasierter <sup>→</sup> wenn Element aus Vektor verwendet  
 → stellt da man Zeiger auftritt und nicht ein Objekt operator[]: zweiter Parameter ganze Zahl, als Index genutzt. Operator gibt Ergebnis auf Pointe zurück unter dem im Index in Vektor Lager gespeichert ist.

- Ware\* Ware Lager:: operator[] (int index) const {
 return lager[index]; }

## Angegebene Länge Vektor:

- int Ware Lager:: belegteLagernplätze() & return lager.size();
- Aufgabe 5: prüfen ob das Element auf Lagerindex zeigt, Wenn ja MHD prüfen. Wenn kleiner als Datum heute (statische Methode), Produkt aus Lager beiden
- void entferneAbgelaufeneWaren (Ware Lager & lager) {
 Ware\* ware;
 for (int i = 0; i < lager.belegteLagernplätze(); i++) {
 ware = lager[i];
 Lebensmittel \*lebensmittel = dynamic\_cast< Lebensmittel \*> (ware);
 if (lebensmittel != nullptr) { → wenn Lebensmittel
Datum mhd = Lebensmittel->getMHD();
 if (mhd < Datum:: heute()) { → MHD vorl. heute
lager.erase (lebensmittel); } } }

## Vektor:

- allgemeine Schreibweise:  
 ↳ vector < Typ > name; vector < Ware\* > lager;  
 → Zugriff v. front() <→ 1. El. v. back() <→ n. El.  
 → v. at (2) oder v[2] → Zugriff auf 3. El.  
 → pop\_back() → enttl. 1. El.  
 → insert() → legt El. an Stelle hinein

## Aufgabe 2: Exceptions: Falls Tier Tiger ist soll Ausnahme geworfen werden.

- wird Wertezuverlässigkeit (Tier &t) { → Bereich w. static cast
if (dynamic\_cast< Tiger\*> (t)) { → wahrscheinlich jmd
else { Tiger \* tiger = dynamic\_cast< Tiger\*> (t); }
Pointe auf Tiger geprüft → ist (tiger != nullptr) { throw "Tiger"; }
else { throw string ("unbekanntes Tier"); } }
try { → prakt. Wertezuverlässigkeit (\*tptr); ← try-blöck
catch (wildes Tier) { throw; } } → catch
catch (string s) { cout << s << endl; }
catch (...) { cout << "unbekannte Ausnahme" << endl; }

## Exception Handling:

- erweitern Implementierung Funk. Werte zuverlässiger
- 2 Dinge prüfen, bevor hinzugefügt wird → max Anzahl Zeichenreich erlaubt?
  - std::exception out-of-range entfernen
  - falls Start!=Endpunkt → std::exc. invalid\_argument
- § 6 if (teilstrichen.size() > maxanzahl) {
 throw out-of-range ("Zu viele Strichen"); }
 if (teilstrichen.back()) → getZiel() → Es wird nicht auf true setzen
! = Strecke → getStartort() { → zweitlich inner string
throw invalid\_argument ("Ziel != Start"); }
- Ergänzen des Outleades mit try block und catch
- out-of-range → Meldung anzeigen, Schleife fortsetzen, es reicht auf true setzen
- invalid\_argument → Meldung ausg., Schleife fortsetzen
- andere Ausnahme, Meldung anzeigen, Ausnahme weitergereicht werden an Aufrufer der Funktion plausiabel()
- try { → mit Parameter, der an what() weisungsgültig war
catch (out-of-range & oor) {
 cout << oor.what() << endl;
 es\_right = true; } → what() aus Ad. exception
}
- catch (invalid\_argument & ia) {
 cout << ia.what() << endl; }
- catch (...) {
 cout << "unvorhergesehener Fehler" << endl;
 throw; } → Fehler der man nicht gecatcht hat wird wieder geworfen, wenn einfach nur throw