**12. Operator Overloading**

**Ex. 1** class Alfa{

unsigned char c;

public: Alfa(unsigned char bra= 1):c(bra){}

//Prefix version

const Alfa& operator++(){

cout << "++Alfa\n";

c++; return \*this; }

//postfix version

const Alfa operator++(int){

cout << "Alfa++\n";

Alfa vor(c);

c++; return vor; } };

**Ex. 2, 3, 4**

class Alfa{

unsigned int c;

public:

Alfa(unsigned int bra= 1):c(bra){}

Alfa operator+(const Alfa& right) const {

return Alfa(c +right.c); }

Alfa operator-(const Alfa& right) const {

return Alfa(c -right.c); }

const Alfa& operator++(){ //Prefix

cout << "++Alfa\n";

c++;

return \*this; }

const Alfa operator++(int){ //Postfix

cout << "Alfa++\n";

Alfa Vor(c);

c++;

return Vor; }

const Alfa& operator--(){ //Prefix

cout << "--Alfa\n";

--c;

return \*this; }

const Alfa operator--(int){ //Postfix

cout << "Alfa--\n";

Alfa Vor(c);

c--;

return Vor; }

void print(){ //Refering to p.442

cout << "0x" << std::hex << int(c) <<

std::dec << endl; } };

void f\_1(Alfa c) {

Alfa\* ap = &c;

++c;

c++;

c.print();

--c; c.print(); c--; }

int main() {

Alfa ora; f\_1(ora);

**Ex. 5**

Alfa& operator++(){ //Prefix

cout << "++Alfa\n";

c++;

return \*this; }

Alfa& operator--(){ //Prefix

cout << "--Alfa\n";

--c;

return \*this; } // The postfix versions don’ t change

**Ex. 6**

ostream& operator<<(ostream& cout , const Alfa c){

cout << c.c << endl;

return cout; }

int main() {

Alfa ora; cout << ora;

**Ex. 7**

friend const Alfa operator+(const Alfa& links, const Alfa& rechts); //Declarations

friend const Alfa operator-(const Alfa& links, const Alfa& rechts); // In the class body

const Alfa operator+(const Alfa& links, const Alfa& rechts) {

return Alfa (links.c + rechts.c); }

const Alfa operator-(const Alfa& links, const Alfa& rechts) {

return Alfa(links.c - rechts.c); }

**Ex. 8**

friend const Alfa operator-(const Alfa& C); //Class declaration

const Alfa operator-(const Alfa& C) {

cout << "-Alfa\n";

return Alfa(-C.c); }

int main() { Alfa Katrina(3); (-Katrina).print();

**Ex. 9**

class Alfa{

char se;

public: Alfa( char bra):se(bra){}

friend ostream& operator<<(ostream& os, const Alfa& c);

friend istream& operator>>(istream& is, Alfa& c); };

ostream& operator<<(ostream& os , const Alfa& c){

os << c.se << endl;

return os; }

istream& operator>>(istream& is, Alfa& B) {

is >> B.se ;

return is; }

int main() {

Alfa Katrina('W');

cin >> Katrina;

cout << Katrina;

**Ex. 10** Perhaps this requires to access the function operator--(int)

Argument by accessing the addresses in the stack as described in chapter

11. At one position is the address of the function stored and on the next

maybe its arguments are stored.

**Ex. 11** class Nummer{

double egt; public:

Nummer(double tg = -9):egt(tg){} ~Nummer(){}

const Nummer operator+(const Nummer& recht)

const { return Nummer(egt + recht.egt); }

const Nummer operator-(const Nummer& recht)

const { return Nummer(egt - recht.egt); }

const Nummer operator/(const Nummer& recht)

const { return Nummer(egt / recht.egt);}

const Nummer operator\*(const Nummer& recht)

const { return Nummer(egt \* recht.egt);}

Nummer& operator=(const Nummer& recht)

{ egt = recht.egt;

return \*this; }

void print(){

cout << egt << endl; } };

void h(){

Nummer Nm\_1(4), Nm\_2, Nm\_3;

Nm\_3 =Nm\_1 + Nm\_2; Nm\_3.print();

Nm\_3 =Nm\_1 - Nm\_2; Nm\_3.print();

Nm\_3 =Nm\_1 / Nm\_2; Nm\_3.print(); }

int main() { h();

**Ex. 12** **Return value optimization ?**

Ex. 13

class Nummer{

double\* egt; public:

Nummer(double\* tg ):egt(tg){}

Nummer& operator=(const Nummer& Recht){

if(this == &Recht) return \*this;

\*egt =\*Recht.egt;

return \*this; }

void print(){

cout << egt << endl; } };

int main() {

double ro;

double& T = ro;

Nummer NM\_1(&ro), NM\_2((&ro +2));

NM\_1.print(); NM\_2.print();

NM\_1 = NM\_2; NM\_1.print();

cout << &ro<<"\n";

**Ex.** **14**

class Vogel{

static int i;

string sig; public:

Vogel(string s ):sig(s +" Vogel #"+to\_string(i)) {++i;}

~Vogel(){}

Vogel(const Vogel& Vog):sig(Vog.sig) { cout<<\*this << Vog; }

friend ostream& operator<<(ostream& os, const Vogel& v) {

return os <<v.sig << endl; }

Vogel& operator=(const Vogel& v) {

if(this ==&v) return \*this;

sig = v.sig; return \*this; } };

int Vogel::i= 1;

int main() {

Vogel Artur("Artur"), Kamila("Kamila"), Sasa("Sasa"), Tonia("Tonia");

//cout << Sasa << Tonia;

Vogel Helga = Tonia;

**Ex. 15**

class VogelHaus{

Vogel V1; Vogel\* V1p; Vogel& V1r;

VogelHaus& operator=(const VogelHaus& VH);

VogelHaus(const VogelHaus&); public:

VogelHaus(Vogel Vgl, Vogel\* Vglp, Vogel& Vglr):V1(Vgl), V1p(&V1),

V1r(V1){}

friend ostream& operator<<(ostream& os, const VogelHaus& VH){

return os << " Dieses VogelHaus enthaelt: " << VH.V1r; } };

int main() { Vogel Tucan("Tula");

VogelHaus VH(Tucan, &Tucan,Tucan);

**Ex. 16**

//Inside Vogel class Int uri; public:

Vogel(string s, int a ):sig(s +" Vogel #"+to\_string(i)+" "),uri(a) {++i;}

friend ostream& operator<<(ostream& os, const Vogel& v) {

return os << v.sig << v.uri<< endl;

const Vogel operator\*(const Vogel& A) const {

return Vogel("",(uri \* A.uri)); }

//VogelHaus constructor VogelHaus(Vogel Vgl, Vogel\* Vglp, Vogel& Vglr, int a)

:V1(Vgl), V1p(&V1), V1r(V1), ora(a){}

friend ostream& operator<<(ostream& os, const VogelHaus& VH){

return os << " Dieses VogelHaus enthaelt: " << VH.ora; }

const VogelHaus operator/(const VogelHaus& Recht) const {

return VogelHaus(V1, V1p,V1r,(ora /Recht.ora)); }

int main() { Vogel Tucan("Tula",13), Stasa("Sasa",3); cout <<(Tucan \*Stasa);

VogelHaus VH(Tucan, &Tucan,Tucan,43), VH2(Tucan, &Tucan,Tucan,241);

cout <<(VH2/VH);

Ex. 17

**Ex.** **18** bool operator--() { //Prefix

if(index <= oc.a.size()) return false;

if(oc.a[--index] == 0) return false;

return true; }

bool operator--(int) { //Postfix

return operator--(); //Use prefix version }

**Ex. 19** class Fee { public:

Fee(int) {cout << "Fee Beschafft!\n"; }

class Fo { int i; public:

operator Fee() const { cout << "Fo Kopierung!\n"; return Fee(i); }

**Ex.** **20**

class Alfa{ public: friend Alfa operator=( Alfa&, const Alfa& );

};

Alfa operator=(Alfa& af4, const Alfa& af8) {

if (this == &af8) return \*this;

ur =af8.ur;

return \*this; } // Compiler error:

Alfa operator=(Alfa&, const Alfa&)’ must be a nonstatic member function!

**Ex. 21**

class Alfa{ public: int ur;

Alfa(int a):ur(a) {} };

class Bravo{int gor =-3;

public: //Bravo(int) {}

operator Alfa() const {string sc="CC op =Anruf.";

cout << sc;

return Alfa(gor); } };

Alfa f1(Alfa af1, Bravo Bra1) {

af1=Bra1;

cout << af1.ur;

return af1; }

**Ex. 22**

class DogHouse {

string houseName;

public:Dog\* p; // Make the Dog pointer public.

In main add the followin to see the addresses of fidos, fidos2

cout << fidos.p;

cout << fidos2.p<<endl;

**Ex.23** Same as exercise 14

**Ex. 24** class Alfa { string sn; public:

Alfa(const string ng):sn(ng){cout << "Alfa obj\n";}

void print(){ cout <<sn<< endl; } };

class Lucia{

Alfa fa1(string s="hertor");

public: Lucia(){

cout <<" Lucia"; }

void print(){ cout <<fa1.s<< endl; } };

int main() { Lucia Luc;

//Marc = Luc; Lucia Jon = Luc;

**Ex. 25, 26** Trivial

**Ex.27** Pending!

#include <iostream>

#include <vector>

using namespace std;

class Obj {

static int i, j;

public:

void f() { cout << i++ << endl; }

void g() { cout << j++ << endl; }

};

// Static member definitions:

int Obj::i = 47;

int Obj::j = 11;

// Container:

class ObjContainer {

vector<Obj\*> a;

public:

void add(Obj\* obj) { a.push\_back(obj); }

class SmartPointer;

friend class SmartPointer;

typedef void (ObjContainer::\*PMF)();

class SmartPointer {

public: class Obj;

friend class Obj;

// typedef void (ObjContainer::\*PMF)();

private:

ObjContainer& oc;

unsigned int index;

PMF pmem;

public:

SmartPointer(ObjContainer& objc, PMF pmf) : oc(objc),

pmem(pmf) {

index = 0;

}

// Return value indicates end of list:

bool operator++() { // Prefix

if(index >= oc.a.size()) return false;

if(oc.a[++index] == 0) return false;

return true;

}

bool operator++(int) { // Postfix

return operator++(); // Use prefix version

}

void operator() () {

return (oc.a[index]->\*pmem)();

}

/\* Obj\* operator->() const {

require(oc.a[index] != 0, "Zero value "

"returned by SmartPointer::operator->()");

return oc.a[index];

}\*/

};

ObjContainer operator->\*(PMF pmf) {

return ObjContainer(this, pmf);

}

// Function to produce a smart pointer that

// points to the beginning of the ObjContainer:

SmartPointer begin() {

return SmartPointer(\*this);

}

};

int main() {

const int sz = 10;

Obj o[sz];

ObjContainer oc;

for(int i = 0; i < sz; i++)

oc.add(&o[i]); // Fill it up

/\*ObjContainer::SmartPointer sp = oc.begin();

do {

sp->f(); // Pointer dereference operator call

sp->g();

} while(++sp);\*/

}

**Ex. 28**

class Orange {public: Orange(){} };

class Apfel { public:

explicit Apfel(Orange) {} };

void g(Apfel) {

cout << "Der Apfel"; }

int main(){

Orange Ran;

g(Apfel(Ran));

**Ex. 29** // Inside the class: friend const Number operator\*(const Number&,

const Number&);

//After the class body: const Number operator\*(const Number& n1,

const Number& n2) {

return Number(n1.i \* n2.i); }

**Ex. 30**  **Non functional!**

class Num2{ int y;

public:

Num2(int iot=0):y(iot) {}

class Num1;

friend class Num1;

// operator Num1() { return Num1(y); }

friend Num2 operator+(const Num1&, const Num2&);

void print() {

cout<< y<< endl; } };

class Num1 { int i;

public: Num1(int a):i(a) {}

operator Num2() const { return Num2(i); }

friend Num2 operator+(const Num1&, const Num2&); };

Num2 operator+(const Num1& n1, const Num2& n2){

return Num2(n1.i + n2.y); }

int main() { Num1 n(-3);

Num2 n2(7);

(n + n2).print();

**Ex.** **32**  Assembly code

**13. Dynamic Object Creation**

Ex. 1, 2

class Gezaehlt { int id;

static int zaehlen; public:

Gezaehlt():id(zaehlen++) {

cout << "id: " <<id<< endl; }

~Gezaehlt(){ cout <<id << " Zerstoerung \n"; }

};

int Gezaehlt::zaehlen = 0;

int main() {

cout <<"Vor Beschaffen!\n";

Gezaehlt \*Afa1 = new Gezaehlt;

cout <<"Zwischen Beschaffen und Zerstoerung!\n";

delete Afa1;

cout << "Nach Zerstoerung!\n";

**Ex. 3**

int main() { PStash GezaehltSta;

for(int i = 0; i < 3; i++)

GezaehltSta.add(new Gezaehlt);

for(int k = 0; k < GezaehltSta.count(); k++)

delete GezaehltSta.remove(k);

**Ex.4, 5 Cannot use class’ vector add function!**

**Ex. 6** class Gezaehlt {

static int zaehlen; public: int id;

Gezaehlt():id(zaehlen++) {

cout << "id: " <<id<< endl; }

~Gezaehlt(){ cout <<id << " Zerstoerung \n"; }

void f() { cout<< "f Funktion!\n"; } };

int Gezaehlt::zaehlen = 1;

int main() { PStash GezaehltSta;

for(int i = 0; i < 3; i++)

GezaehltSta.add(new Gezaehlt);

for(int j = 0; j < GezaehltSta.count(); j++) {

cout << "GezaehltSta[" << j << "] = "

<< ((Gezaehlt\*)GezaehltSta[j])->id << endl;

((Gezaehlt\*)GezaehltSta[j])->f(); }

for(int k = 0; k < GezaehltSta.count(); k++)

delete GezaehltSta.remove(k);

**Ex. 7** **Pending!**

**Ex. 8** int main() {

Gezaehlt\* Gzp = new Gezaehlt[2];

delete Gzp;

Errormessage: Error in `/home/a.out': munmap\_chunk(): invalid pointer: 0x00000000013c4c28 \*\*\*    Aborted

**Ex. 9**

int Gezaehlt::zaehlen = 1;

int main() { Gezaehlt\* Gzp = new Gezaehlt;

cout << Gzp->id <<endl; //Prints 1

(void\*)Gzp;

delete Gzp;

cout <<Gzp->id; //Prints 0

**Ex. 10** NewHandler.cpp

**Ex. 11**

class Alfa{ enum { sz =3};

public:

void\* operator new(size\_t sz){

cout <<"New operator single object\n";

return ::new char[sz]; }

void\* operator new[](size\_t sz){

cout <<"New operator \n";

return ::new char[sz]; }

void operator delete[](void\* p) {

cout<< "Delete \n";

::delete[]p; } };

int main() { Alfa\* A= new Alfa[3];

Alfa\* As = new Alfa;

delete []A;

**Ex.** **12**

**Ex. 13** Without the bad\_alloc the compiler creates objects at runtime unceasingly

class NoMemory {

enum { sz = 10 };

int a[sz]; ...

int main() {

static int zahlen = 0;

while(1){ NoMemory\* nm = new NoMemory;

/\*catch(bad\_alloc) {

cerr << "Out of memory exception" << endl;

}\*/

cout << "nm = " << zahlen << endl;

zahlen++; }\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ //Throwing bad alloc

int main() { static int zahlen = 0;

try { NoMemory\* nm = new NoMemory;

cout << "nm = " << zahlen << endl;

zahlen++; }

catch(bad\_alloc) {

cerr << "Out of memory exception" << endl; } //The compiler prints:

NoMemory::operator new

Out of memory exception

**Ex.14**

The string argument of new must be placed in the static vector<string> sve. Placement New(size\_t, string sr) is a

class overloaded function. Sve.add(&sr) is not recognized as method of the class std::vector>.

class Bravo { static vector<string> sve;

public:

Bravo(){cout <<"Konstruktor! \n";}

~Bravo(){cout << "Zerstoerer! \n";}

void\* operator new(size\_t, string sr) { sve.add(&sr);

//**Does not recognize add() !** } }; //**How to pass the string argument of new to class member static vector<string> ?**

int main() { Bravo\* Bv = new("Soares") Bravo;

delete Bv;

**Ex. 15** **Incomplete! Vector class non functional**

class Widget {... public:

static int zahlen; ... };

int Widget::zahlen = 1;

class GeheugenKontroller{ public:

Widget\* Wgtp;

GeheugenKontroller(){}

~GeheugenKontroller() { cout << Wgtp-> zahlen <<" Destructeur appele" << endl; }

};

int main() {

GeheugenKontroller B1;

Widget\* a1= new Widget[7];

//delete []a1;

B1.Wgtp = a1;

**14. Inheritance – Composition**

**Ex. 1** class Vehicle{ public:

Vehicle () {cout<< "Vehicule\n";}

Engine engine;

Wheel wheel[4];

Door left, right;

void boost(){cout << "Boost!\n";}

void Turbo() {cout <<"Turbocharging activated!\n"; }

void Intercooler() {cout << "Intercooler activated\n"; }

void Kompressor() {cout << "Kompressor for turbocharging activated!\n"; }

};

class Car: public Vehicle {

public:

Car() : Vehicle(){cout <<"Caro\n"; } };

**Ex. 2** class A {

public:A() { cout <<"A Konstruktion!\n"; } };

class B {

public: B() {cout <<"Beta Konstruktion \n"; } };

class C : public A{ B beta; };

**Ex. 3** class A {

public:A(){ cout <<"A Konstruktion!\n"; }

~A(){cout <<"C Zerstoerung!\n"; } };

class B: public A {

public: B() {cout <<"Beta Konstruktion \n"; }

~B(){cout <<"C Zerstoerung!\n"; } };

class C : public B{ public: C() {cout << "C Konstruktion\n"; }

~C(){cout <<"C Zerstoerung!\n"; } };

**Ex. 4** class D : public C {

C c\_uno;

public: D(int ii):c\_uno(2), C(ii) { cout << "D Kons\n"; }

~D() {cout << "D Zerst\n"; } };

int main() { D c(7);

**Ex. 5** class D : public B {

C c\_uno;

public: D(int ii):c\_uno(2),B(ii) { cout << "D Kons\n"; }

~D() {cout << "D Zerst\n"; }

};

Main() { D c(-7);

**Ex. 6** class Derived3 : public Derived2 {

Member4 m3\_4;

Member5 m3\_5;

public:

Derived3() : m3\_4(1), Derived2(), m3\_5(3) {

cout << "Derived3 constructor\n"; }

~Derived3() {

cout << "Derived3 destructor\n"; } };

**Ex. 7**

int main() {

Derived2 d2;

x = d2.f(); // The Base class f() is hidden

x = d2.Base::f(); // Base class function f is called with explicit scope resolution.

Derived4 d4;

x = d4.f(); // Base class f hidden.

x = d4.Base::f(); // Similarly here.

**Ex. 8** class Base {public:

int h() const { cout << "Base::h()\n";

return 1; }

int h(string) const { return 1; }

int h(int) const { cout << " Base::h drei"; return 1; } };

class Derived3 : public Base {

public: // Change return type:

void h() const { cout << "Derived3::h()\n"; }

**Ex.9** **Pending! Use of vector class functions**

**Ex. 10** class Base {long lona;

public: Base():lona(455) {}

void print() { cout << lona<< endl; }

**Ex. 11** Sample solution. Not according to the exercise specification!

class Asteroid:public PStash {

long lona;

public: Asteroid():lona(455) {}

int add(Asteroid\* element) {

PStash::add(element); }

Asteroid\* operator[](int index) const {

return (Asteroid\*)PStash::operator[](index); }

Asteroid\* remove(int index) {

return (Asteroid\*)PStash::remove(index); }

};

int main() {

Asteroid Astra;

for(int i = 0; i < 25; i++)

Astra.add(new Asteroid);

for(int j = 0; j < Astra.count(); j++)

cout << "Astra[" << j << "] = "

<< Astra[j] << endl;

// Clean up:

for(int k = 0; k < Astra.count(); k++)

delete Astra.remove(k);

**Ex. 12** **Vector class functions**

**Ex. 13** class Chess : public Game {public:

Chess() {cout << "Echecs default\n"; }

Chess(Chess& Ech): Game(Ech){cout << "Echecs copy-consr\n"; }

Chess& operator=(const Chess& c) {

Game::operator=(c);

cout << "Echecs::operator=()\n";

return \*this; } };

**Ex. 14** class Reiser { string sn;

public: Reiser(string Rsn): sn(Rsn) { cout <<

"Reiser Konstruktor\n"; }

Reiser(const Reiser&) { cout <<"Reiser Kopierung\n"; }

Reiser& operator=(const Reiser& Raser) {

sn = Raser.sn;

return \*this; } };

class Pager { string sn;

public: Pager(string Rsn): sn(Rsn) { cout << "Pager Konstruktor\n"; }

Pager(const Pager&) { cout <<"Pager Kopierung\n"; }

Pager& operator=(const Pager& Raser){

sn = Raser.sn;

return \*this; } };

class GeschaftsReiser: public Reiser { Pager pago;

public:

GeschaftsReiser():Reiser(""),pago("") { cout << "GeschaftsReiser Kons\n"; }

GeschaftsReiser(string Rsn): Reiser(Rsn),pago(Rsn) { cout <<

"GeschaftsReiser Konstruktor\n"; }

GeschaftsReiser(const GeschaftsReiser& Raser):Reiser(Raser),pago(Raser.pago) { cout

<<"GeschaftsReiser Kopierung\n"; }

GeschaftsReiser& operator=(const GeschaftsReiser& Raser){

pago = Raser.pago;

Reiser::operator=(Raser);

return \*this; } };

**Ex. 15** class Alfa {

public: Alfa(){}

static void fina() {

cout << "Fina\n"; }

static void Tina() {

cout << "Tina \n"; } };

class Bravo: public Alfa {

public:

static void fina() {

cout << "Bravo Fina \n"; } };

**Ex. 16**

**Ex. 1**7 class Alfa {

public: Alfa(){}

static void fina() {

cout << "Fina\n"; }

static void Tina() { cout << "Tina \n"; } };

class Bravo: private Alfa { public:

void spielen(Alfa& a) {

a.fina(); a.Tina(); } };

class Zito: protected Alfa {

public: Zito() {}

void spielen(Alfa& a) {

a.fina(); a.Tina(); } };

int main() { Zito Zi; //The base class remains inaccessible for both of the

Zi.spielen(Zi); two derived classes!

**Ex. 18** class Derived : public Base {

public:

void Access() { cout <<read(); }

**Ex. 19**

error: ‘int Base::value(int) const’ is inaccessible

**Ex. 20**

class Spaceship { public: void fly() { }

};

class Shuttle : public Spaceship {

public: void land(Spaceship\*) { cout <<

"Shuttle landing! \n"; } };

int main() { Shuttle Sh3294;

Shuttle\* Shpo;

Sh3294.land(Shpo);

**Ex. 21**

class Instrument { public:

void vorbereiten(note) { cout << "Vorbereitung \n" ; }

void tune(Instrument& i) {

// ... i.play(middleC);

i.vorbereiten(Cflat);

**Ex. 22** Without the virtual keyword the program calls the Instrument’s

play method. By putting the virtual keyword before the play method in

Instrument class it is the Wind’s play method that is called.

**Ex. 23** class Parent {...

Parent& operator=(const Parent& Kn) { // Addition of operator= in Parent class

i=Kn.i;

return \*this; }

class Child {... Child& operator=(const Child& Kn) { Addition of operator= in Child class

Parent::operator=(Kn);

i=Kn.i;

m = Kn.m;

return \*this; }

class GrossKind : public Child {

Member m;

public: GrossKind(int i):Child(i),m(i) {}

GrossKind(const GrossKind& Kn):Child(Kn), m(Kn.m){}

GrossKind& operator=(const GrossKind& Kn) {

Child::operator=(Kn);

m = Kn.m;

return \*this; }

friend ostream&

operator<<(ostream& os, const GrossKind& c){

return os << (Child&)c << c.m << "GrossKind: " <<endl;

}

**Ex. 24** class Child : public Parent {

Child(const Child& chd): i(chd.i), m(chd.m){cout <<"Chd cc\n";}

int main() { Child c(2);

Child c2 = c;

cout << c2; Parent: 0              //The parent class was not initialized

Member: 2             correctly

Child: 2

Solution: class Child : public Parent {...

Child(const Child& chd):Parent(chd), i(chd.i), m(chd.m) {cout <<"Chd cc\n"; }

**Ex. 25,**  **Vector class functions!**

**Ex. 26** class Rock { public:

Rock(){ cout << " Konstr\n"; }

Rock(const Rock& ri) {cout << "cc !\n"; }

Rock& operator=(const Rock& g) {

cout <<"Rock::operator=()\n";

return \*this; }

~Rock(){cout << "Rock Zerst\n"; } };

int main() {

vector <Rock> VRo; //vector<Rock\*> VRo;

for (int i = 0; i<4; i++) { //Rock\* riki = new Rock;

Rock riki;

VRo.push\_back(riki); }

For the versio of vector<Rock\*> VRo

for (int i = 0; i<4; i++) //To destroy the Rock pointers

delete VRo.at(i);

**Ex. 27** **Non functional!**

class Subject { public:

void f(){cout << "F\n"; }

void g(){ cout << "G \n"; }

void h(){ cout << "h \n"; } };

class Im1:public Subject { public:

void f()const {cout << "Im 1F\n"; }

void g() const { cout << "Im 1 G\n"; } };

class Im2:public Subject { public:

void f()const {cout << "Im 2F\n"; }

void g() const { cout << "Im 2 G\n"; } };

class Proxy:public Subject {

Subject\* sutu; public:

Proxy(Subject\* r):sutu(r){}

void hanyu() {sutu->f();}

void liu() {sutu->g(); }

void sen() {sutu->h(); } };

int main() {Subject t; Proxy Lina(&t);

Lina.hanyu();

**Ex.** **28** class Alfa:public Widget{

public: };

int main() { Alfa\* w = new Alfa;

Alfa\* wa = new Alfa[25];

**Ex. 29** **Incomplete!**

The Framis class is written for Framis objects with fixed size. Sizeof(Framis) = 10.

class Alfa: public Framis {

char f;

int a; }; // Class Alfa has size 16. The size should make particular the class only for

Framis objects.

int main() { //cout << sizeof(Framis)<< endl;

cout << sizeof(Alfa);

Alfa\* f[Alfa::psize];

try { for(int i = 0; i < Alfa::psize; i++)

f[i] = new Alfa;

new Alfa; // Out of memory }

catch(bad\_alloc) { cerr << "Out of memory!" << endl; }

delete f[10];

f[10] = 0;

// Use released memory:

Alfa\* x = new Alfa;

delete x;

for(int j = 0; j < Alfa::psize; j++)

delete f[j]; // Delete f[10] OK \*/

**15: Polymorphism**

**Ex. 1**

class Gestalt { public:

virtual void zeichnen() { cout <<"Gestalt zeichnen"<< endl; } };

class Zirkel:public Gestalt { public:

void zeichnen() { cout <<"Zirkel zeichnen"<< endl; } };

class Quadrat: public Gestalt { public:

void zeichnen() { cout <<"Quadrat zeichnen"<< endl; } };

int main() {Gestalt\* Aras = new Gestalt[3];

for (int i=0; i<3; i++)

Aras[i].zeichnen();

**Ex. 2**

Class Gestalt {... public: virtual void zeichnen()=0; ...};

void Gestalt::zeichnen() {cout <<"Gestalt zeichnen"<< endl; }

Gestalt::Gestalt() {zeichnen(); }

int main() { Gestalt Aras; // error: cannot declare variable

// ‘Aras’ to be of abstract type ‘Gestalt’

**Ex.** **3**

void fGe(Gestalt gwerte) {

cout << "fGe"<< endl; //error: cannot declare parameter

‘gwerte’ to be of abstract type ‘Gestalt’ }

void fGe(Gestalt& gwerte) {

cout << "fGe"<< endl;

}

int main() { Quadrat Aras;

fGe(Aras); // prints Gestalt zeichnen \n fGe

**Ex**. 4

class A {

virtual void f() const {cout << "A::f() \n";} };

class C : public B, public A {

void f() const { // Redefinition cout << "C\n"; } };

void fossa(A& al) {

al.f(); }

int main() { C c(47);

fossa(c);

**Ex. 5**

class Instrument {

virtual void vorbereiten(){cout << "Vorbereiten Instru\n"; }...

class Wind : public Instrument { public:

void vorbereiten(){cout << "Vorbereiten Wind\n"; }...

void tune(Instrument& i) { // ...

i.play(middleC);

i.vorbereiten(); }

**Ex. 6**

class Rodent { public:

virtual void fernando() { cout << "Rodent \n"; } };

class Gerbil: public Rodent {public:

void fernando() { cout <<"Gerbil \n"; } };

class Hamster: public Rodent { public:

void fernando() { cout << "Hamster\n"; } };

int main() { Rodent\* alixa[3];

Gerbil jorj;

Hamster Tomi;

alixa[0] =&jorj; alixa[1]= &Tomi; alixa[2]= &jorj;

for (int i=0; i<3; i++)

alixa[i]->fernando();

**Ex.** **7** **Incomplete. Vector class functions!**

int main() { vector<Rodent\*> Sasa;

Gerbil\* jorj;

Hamster\* Tomi;

Sasa.push\_back(dynamic\_cast<Rodent\*>(jorj));

Sasa.push\_back(dynamic\_cast<Rodent\*>(Tomi));

cout<<Sasa.size()<< endl;

Rodent\*\* aris = Sasa.data();

for (unsigned i=0; i<Sasa.size(); ++i)

aris[i]->fernando();

**Ex. 8**

class BlauHamster: public Hamster {public:

void fernando() { cout << "Blau Hamster\n"; } };

int main() { Gerbil Tomi; Hamster jorj; BlauHamster Hulio;

Rodent\* alixa[3];

alixa[0] =&jorj; alixa[1]= &Tomi; alixa[2]= &Hulio;

for (int i=0; i<3; i++)

alixa[i]->fernando();

**Ex. 9**

//Before virtual destructor

int main() { Rodent\* jorj = new Hamster;

delete jorj; // Prints: Rodent Zerstoerer!

**virtual** ~Rodent() {cout << "Rodent Zerstoerer! \n"; }

// Prints: Hamster Zerstoerer! Rodent Zerstoerer!

**Ex.10** class Rodent { public:

virtual ~Rodent() {cout << "Rodent Zerstoerer! \n"; }

virtual void fernando() const = 0; };

**Ex. 11**

Same almost structure as ex. 7. Need to use vector class functions.

Must be able first to do ex. 7.

**Ex. 12**

**Ex.13**

virtual string speak()=0;

**Ex. 14**

class Pet {

virtual string name() const=0;

virtual string speak() const=0;

**Ex. 15**

class Aza { public://Aza () {}

virtual void fina() {cout << "Aza\t"; } };

class bravo : public Aza { public:

bravo() {Aza::fina(); cout <<"bravo Konstru\n";}

void fina(Aza& fzi) {fzi.fina();

cout << "Kinderklass\n"; } };

int main() {bravo vissimo;

// Alternative AlAza\* zoi = new bravo; vissimo.fina(\*zoi);

Aza zoi; vissimo.fina(vissimo);

**Ex. 16**

class Derivatif: public Derived { public:

void f() { cout << "Derivatif\n"; }

~Derivatif(){ cout <<"Derivatif Zerstoerer!\n"; } };

int main() {

Base\* bp = new Derivatif; // Upcast

delete bp;

**Ex. 17**

class Base {

virtual ~Base() { cout << "Base1()\n";

f(); }

class Derived : public Base {

~Derived() { cout << "~Derived()\n";

f(); }

class Derivatif: public Derived { public:

~Derivatif(){ cout <<"Derivatif Zerstoerer!\n";

f();} //Each class destructor is calling its class’ f version

**Ex.18**

class Primera {short juri; };

class Segunda :public Primera {

short frida; };

void fina(Primera piea) {

cout << sizeof(piea) <<endl; }

int main() {//Primera sia; /\*The size of the base is 2 (one short member)

Segunda feira; The size of the derived is 4. When the function is called with

cout << sizeof(feira)<< endl; Argument an object of the derived class it

fina(feira); upcasts it to a base object thus its size reduces from4 to 2. \*/

**Ex. 19** Assembly code generation

**Ex. 20** **Incomplete. Requires clock function from <ctime> class.**

class Primera {short juri;public:

void fnvirtus() {cout << "Primera::fnvirtus\n"; }

virtual void fvius() { cout <<"Primera::Virtus\n"; } };

class Segunda :public Primera {

void fnvirtus() {cout << "Segunda::fnvirtus\n"; }

void fvius() { cout <<"Segunda::fvius\n"; } };

int main() { Primera\* feira = new Segunda;

feira->fnvirtus(); //Calls Primera version

feira->fvius(); //Calls Segunda version

**Ex. 21**

#define CLASS(ID) class ID { \

**virtual** ~ID() { cout << #ID " destructor\n"; } \

**virtual** void f() { cout << "Virtus Basis\n"; }\

class Derived1 : public Base1 {

void f() { cout << "Derived1 f\n"; }

class Derived2 : public Derived1 {

void f() { cout << "Derived2 f\n"; }

int main() {Base1\* ogre = new Derived2;

ogre->f();

**Ex. 22**

Only the overriden function can be called through the derived class object which is defined

In the derived class body.

After upcasting all the functions can be called the overriden function

of the derived class is called not of the base class.

After removing the overriden definition from the derived class the base class functions cannot be

called through the derived class object.

class Primera {public:

void f1() { cout << "11\n"; }

void f1(char ci) {cout << "22\n"; }

virtual void f1(float f) { cout <<"33\n"; } };

class Segunda: public Primera {public:

//void f1(char ci) {cout <<"Anna\n"; }

//void f1();

void f1(float f) { cout <<"33\n"; } };

int main() { char c; float pi;

Segunda seni;

seni.f1();

seni.f1(c);

**Ex. 23**

class PetFood {

virtual PetFood& eats() = 0; ...

class Bird : public Pet {

PetFood& eats() { return bf; } ...

class Cat : public Pet {

CatFood& eats() { return cf; }

int main() {

for(int i = 0; i < sizeof p / sizeof \*p; i++)

cout << p[i]->type() << " eats "

<< (p[i]->eats()).foodType() << endl;

Cat::CatFood& cf = c.eats();

Bird::BirdFood& bf= dynamic\_cast<Bird::BirdFood&>(b.eats());

**Ex. 24**

When the compiler has an object it knows the exact type and performs early binding.

When pointers or references are involved during upcasting as in Early.cpp addresses

are used and the types of objects are not known. In such cases late binding might be

employed by the compiler.

**Ex. 2**5 **Incomplete.** Does not return the subtypes after upcasting.

class Primera {public:

Primera(){}

virtual Primera\* Klonos(Primera\*);

Primera(const Primera&){cout<<"Cc\n";}

};

Primera\* Primera::Klonos(Primera\* su) {

Primera b = \*this;

Primera\* k= &b;

cout <<"Primera Klonos: "<< (long)&k <<endl;

cout <<"this address "<< (long)this << endl;

return k; }

class Terca:public Primera { public:

Terca() {}

Terca(Terca&){}

Terca\* Klonos(Terca\*); };

Terca\* Terca::Klonos(Terca\* te) {

Terca b = \*this;

Terca\* k= &b;

cout <<"Terca Klonos: "<< (long)&k <<endl;

cout <<"this address "<< (long)this << endl;

return k; }

int main() { Terca trito;

Primera\* primo1 = new Terca;

primo1->Klonos(trito);

/\*Primera primo1;

primo1.Klonos(&primo1);

static\_cast<Primera>(trito);

trito.Klonos(&trito);

**Ex. 26**

**Ex. 2**7

class Matrix;

class Scalar;

class Vector;

class Tensor;

class Math {

virtual Math& multiply(Tensor\*) = 0;

virtual ~Math() {} };

class Tensor : public Math {

public:

Math& operator\*(Math& rv) {

return rv.multiply(this); // 2nd dispatch }

Math& multiply(Matrix\*) {

cout << "Matrix \* Tensor" << endl;

return \*this; }

Math& multiply(Scalar\*) {

cout << "Scalar \* Tensor" << endl;

return \*this; }

Math& multiply(Vector\*) {

cout << "Vector \* Tensor" << endl;

return \*this; }

Math& multiply(Tensor\*) {

cout << "Tensor \* Tensor" << endl;

return \*this; } };

int main() { //Prints: error: cannot declare variable ‘m’ to be of abstract type ‘Matrix’

Matrix m; Vector v; Scalar s, Tensor jik;

**Ex. 28, 29** Assembly language code/documentation

**Ex.** 30

class Alvaro { public:

virtual void Ricardo() {cout << "Base\n"; } };

class Sampaio : public Alvaro {public:

void Ricardo() { cout << "Sampaio esta o Sao Paulo!\n"; } };

void fernanda(Alvaro i) {

i.Ricardo(); }

int main() { Sampaio juliano;

fernanda(juliano); //Whether Ricardo in the base class has the keyword

virtual or not function fernanda with Sampaio argument

always calls the base class Ricardo.

**Ex. 31** Assembly documentation

**Ex. 32** Size and no. of VPTRs.

Ex. 33 Function access VPTR and addresses of an object of a class

**Ex. 34**

void tune(Instrument& i) {

i.play(middleC);

}

void tune(Instrument& i, int g) { //One overloaded version

// ...

switch (g) {

case 1: dynamic\_cast<Wind&>(i).play(middleC); break;

case 2: dynamic\_cast<Percussion&>(i).play(middleC); break;

case 3: dynamic\_cast<Stringed&>(i).play(middleC); break;

case 4: dynamic\_cast<Brass&>(i).play(middleC); break;

} }

int main() { tune(flute,1);

tune(drum,2);

tune(violin,3);

tune(flugelhorn,4);

Ex. 35, 36