**”Thinking in C++”, Bruce Eckel**

**Contents**

**The C in C++ 2**

**Data abstraction 5**

**Initialize & Cleanup 9**

**Function Overloading 11**

**Constants 15**

**Inline Functions 18**

**Name Control 22**

**References & the Copy Constructor 33**

**Operator Overloading 40**

**Dynamic Object Creation 48**

**Inheritance – Composition 51**

**Polymorphism 58**

**Exception Handling 66**

**Strings 68**

**Introduction to Templates 68**

**3: The C in C++ The C in C++**

**Ex. 3** Write a program that uses a while loop to read words from standard input (cin) into a string. This is an “infinite” while loop, which you break out of (and exit the program) using a break statement. For each word that is read, evaluate it by first using a sequence of if statements to “map” an integral value to the word, and then use a switch statement that uses that integral value as its selector (this sequence of events is not meant to be good programming style; it’s just supposed to give you exercise with control flow). Inside each case, print something meaningful.

string ram;

int main() {

int k;

while(true) {

cin >> ram;

if (ram== "Bericht") k=1;

else if (ram=="Ankundigung") k=-3;

else if (ram=="Auftrag") k= -7;

else if (ram =="quitter"){

cout << "Ausgang! \n"; break;}

else k=0;

switch (k){

case 1: cout << "Bericht"<<endl; break;

case -3: cout << "Ankundigung ausgewahlt \n"; break;

case -7: cout << "Auftrag ausgewahlt \n"; break;

default: cout << "Keiner gueltige Wahl \n"; break;

}

} return 0; }

**Ex. 4** Modify Menu.cpp to use switch statements instead of if statements.

int main() {

char c; // To hold response

int k=0;

while(k==0) {

cout << "MAIN MENU:" << endl;

cout << "l: left, r: right, q: quit -> ";

cin >> c;

switch (c) {

case 'q': k=-1; break; // Out of "while(1)"

case 'l': {

cout << "LEFT MENU:" << endl;

cout << "select a or b: ";

cin >> c;

switch(c) {

case 'a':

cout << "you chose 'a'" << endl;

break; // Back to main menu

case 'b':

cout << "you chose 'b'" << endl;

break; // Back to main menu

default:

cout << "you didn't choose a or b!"

<< endl;

break; // Back to main menu

}

break; }

case 'r': {

cout << "RIGHT MENU:" << endl;

cout << "select c or d: ";

cin >> c;

switch(c) {

case 'c': cout << "you chose 'c'" << endl;

break; // Back to main menu

case 'd':

cout << "you chose 'd'" << endl;

break; // Back to main menu

default: cout << "you didn't choose c or d!"

<< endl;

break; // Back to main menu

}break;

} default: cout << "you must type l or r or q!" << endl;

break;}

} cout << "quitting menu..." << endl;

**Ex. 6** Modify YourPets2.cpp so that it uses various different data types (char, int, float, double, and their variants). Run the program and create a map of the resulting memory layout. If you have access to more than one kind of machine, operating system, or compiler, try this experiment with as many variations as you can manage.

short int dosh;

long int dol; float cat; char bird; long double fish;

void f(int pet) {

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

int main() { int i, j, k;

cout << "f(): " << (long)&f << endl;

cout << "dosh: " << (long)&dosh << endl;

cout << "dol: " << (long)&dol << endl;

cout << "cat: " << (long)&cat << endl;

cout << "bird: " << (long)&bird << endl;

cout << "fish: " << (long)&fish << endl;

cout << "i: " << (long)&i << endl;

cout << "j: " << (long)&j << endl;

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

**Ex. 7** Create two functions, one that takes a string\* and one that takes a string&. Each of these functions should modify the outside string object in its own unique way. In main( ), create and initialize a string object, print it, then pass it to each of the two functions, printing the results.

string& refa(string& san) {

san = "Yokosan!";

return san; }

string pola( string\* sut) {

\*sut = "Kirukara!";

return \*sut; }

int main() {string ingi="";

cout << ingi << endl;

cout << pola(&ingi) << endl;

cout << ingi << endl;

cout << refa(ingi);

cout << ingi << endl;

**Ex. 9** Compile and run Static.cpp. Remove the static keyword from the code, compile and run it again, and explain what happens.

void func() {

static int i = 0;

for(int x = 0; x < 10; x++)

cout << "i = " << ++i << endl; }

**Ex. 11** Modify Boolean.cpp so that it works with double values instead of ints.

int main() {

double i, j;

**Ex. 12** Modify Boolean.cpp and Bitwise.cpp so they use the explicit operators (if your compiler is conformant to the C++ Standard it will support

these).

cout << "i && j is " << (i and j) << endl;

cout << "i || j is " << (i or j) << endl;

cout << " (i < 10) && (j < 10) is "

<< ((i < 10) and (j < 10)) << endl;

**Ex. 14** Modify Ifthen.cpp to use the ternary if-else operator (?:).

int main() { int i; while(true) {

cout << "type a number and 'Enter'" << endl;

cin >> i;

i > 5? cout << "It's greater than 5" << endl

: i < 5?

cout << "It's less than 5 " << endl

: cout << "It's equal to 5 " << endl;

cout << "type a number and 'Enter'" << endl;

cin >> i;

i < 10 ?

i > 5 ? // "if" is just another statement

cout << "5 < i < 10" << endl

: cout << "i <= 5" << endl // Matches "if(i < 10)"

:

cout << "i >= 10" << endl; }

**Ch. 4 Data abstraction Ch. 4 Data abstraction**

**Ex. 13** Create an abstract data type that represents a videotape in a video rental store. Try to consider all the data and operations that may be necessary for the Video type to work well within the video rental management system. Include a print( ) member function that displays information about the Video.

#include <iostream>

using namespace std;

struct Video{

string RenterName;

string TapeTitle, RentDate, ReturnDate;

string DurationMinutes;

int RentsNo;

string RentState;

void Initialize(string ClientName, string Title, string Date, string R\_Date, string Duration, int Re\_no, string State);

void print();

};

void Video::Initialize(string ClientName, string Title, string Ren\_Date, string Ret\_Date, string Duration, int Re\_no, string State){

RenterName = ClientName;

TapeTitle = Title;

RentDate = Ren\_Date;

ReturnDate = Ret\_Date;

DurationMinutes = Duration;

RentsNo = Re\_no;

RentState = State;

}

void Video::print( ){

cout << "Holder Name: " <<

RenterName << endl;

cout << "Film Title: " <<

TapeTitle << endl;

cout << "Rent Date: " <<

RentDate << endl;

cout << "Return Date: " <<

ReturnDate << endl;

cout << "Duration: " <<

DurationMinutes << endl;

cout << "Number of Rents: " <<

RentsNo << endl;

cout << "Rent State: " <<

RentState << "\n" << endl;

}

int main()

{

Video Tape1, Tape2;

Tape1.Initialize("Alvaro", "Jurassic Park II",

"23/5/1995", "7/6/1995", "95'", 5, "Out of rent"

);

Tape2.Initialize("Margot", "Twilight",

"11/3/1994", "7/4/1994", "98'", 21, "In rent!");

Tape1.print();

Tape2.print();

// cout << Tape1.TapeTitle;

}

**Ex. 18** Write a function that takes a char\* argument. Using new, dynamically allocate an array of char that is the size of the char array that’s passed to the function. Using array indexing, copy the characters from the argument to the dynamically allocated array (don’t forget the null terminator) and return the pointer to the copy. In your main( ), test the function by passing a static quoted character array, then take the result of that and pass it back into the function. Print both strings and both pointers so you can see they are different storage. Using delete, clean up all the dynamic storage.

#include <iostream>

using namespace std;

struct Klas{

char\* storage;

char\* X (char\* Ch);

char\* fetch(int k);

};

char\* Klas:: X(char\* Ch){

char\* bar = new char[5];

for(int k = 0; k<5; k++)

bar[k] = Ch[k];

storage = bar;

return storage;

}

char\* Klas::fetch(int k){

if (k>= 5)

return 0;

return &storage[k];

}

int main()

{

Klas TVR;

char Ch[] = {'j', 'e', 's','r','y'};

for (int w = 0; w< 5; w++)

char\* Duo = TVR.X(Ch);

int k =0;

char\* cp;

while((cp = TVR.fetch(k++)) != 0)

cout << k << "\t cp " << cp << endl;

cout << TVR.fetch(3);

delete []TVR.storage;

cout << "\n After deletion of storage: " << TVR.fetch(1);

//char\* Tre = TVR.X(Duo);

/\*

char ch[] = {'j', 'e', 's','r','y'};

char\* cp = ch;

int k = 0;

for (int w = 0; w< 5; w++){

// cout << ch[w] <<endl;

cout << cp[w] << endl;

}\*/

return 0;

}

P.285-286

#include <iostream>

#include <cstring>

using namespace std;

const int grose = 20;

struct Hoelderling{

public:

void initialize();

struct Pointer;

friend Pointer;

struct Pointer{

void initialize(Hoelderling\* h);

void next();

void previous();

void top();

void end();

int read();

void set(int i);

private:

Hoelderling\* h;

int\* p;

};

private:

int a[grose];

};

void Hoelderling::initialize(){

memset(a,0, grose\*sizeof(int));

}

void Hoelderling::Pointer::initialize(Hoelderling\* rv){

h = rv;

p = rv->a;

}

void Hoelderling::Pointer::next(){

if(p< &(h->a[grose-1])) p++;

}

void Hoelderling::Pointer::previous(){

if(p > &(h->a[0])) p--;

}

void Hoelderling::Pointer::top(){

p = &(h->a[0]);

}

void Hoelderling::Pointer::end(){

p = &(h->a[grose-1]);

}

int Hoelderling::Pointer::read(){

return \*p;

}

void Hoelderling::Pointer::set(int i){

\*p = i;

}

int main(){

Hoelderling h;

Hoelderling::Pointer h1, h2;

int i;

h.initialize();

h1.initialize(&h);

h2.initialize(&h);

for(i = 1; i<= grose; i++){

h1.set(i);

h1.next();

}

h1.top();

h2.end();

for(i = 0; i <= grose; i++){

cout << "h1 = " << h1.read()

<< ",\t h2 = " << h2.read() << endl;

h1.next();

h2.previous();

}

}

**Ch.6 Initialize & Clean-up Ch.6 Initialize & Clean-up**

Write a simple class with a constructor that prints something to tell you that it’s been called. In main( ) make an object of your class. Add a destructor that prints out a message to tell you that it’s been called. Modify the class so that the class contains an int member. Modify the constructor so that it takes an int argument that it stores in the class

member. Both the constructor and destructor should print out the int value as part of their message, so you can see the objects as they are created and destroyed.Demonstrate that destructors are still called even when goto is used to jump out of a loop.

#include <iostream>

using namespace std;

class Dielectric

{

public: Dielectric();

int Objk1;

~Dielectric();

};

Dielectric::Dielectric()

{

Objk1 = 2;

cout<< "\n Constructor invoked! Objk1 = " <<

Objk1 << endl;

}

Dielectric::~Dielectric(){

cout<< "\n Destructor called! Objk1 = " << Objk1;

}

int main()

{

{

cout<<"\n Vor Objekt Schaffen."<< endl;

Dielectric fn;

cout<<"\n Nach Objekt Beschaffen."<< endl;

for (int k=1; k<3; k++){

goto Salto1;

cout<< k << endl; //Niets ausgedrückt.

}

}

Salto1:

cout<<"\n Nach Objekt Zerstörung."<< endl;

**Ex. 5** Write two for loops that print out values from zero to 10. In the first, define the loop counter before the for loop, and in the second, define the loop counter in the control expression of the for loop. For the second part of this exercise, modify the identifier in the second for loop so that it as the same name as the loop counter for the first and see what your compiler does.

int ghe = 0;

int a = 0;

for(ghe = 0; ghe<11; ghe++)

cout << a++ << "\t";

cout <<"\n" << &ghe <<endl;

int b = 0;

for (int ghe =0; ghe<11; ghe++){

if (ghe == 6)

break;

cout << b++ << "\t" << endl;

}

int\* r = &ghe;

cout <<"\n"<< r<<"\t" << \*r;

}

**Ex. 7** Use aggregate initialization to create an array of double in which you specify the size of the array but do not provide enough elements. Print out this array using sizeof to determine the size of the array. Now create an array of double using aggregate initialization and automatic counting. Print out the array.

int main()

{

double vetor[3] = {7.0};

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

cout << vetor[i] << "\n" <<endl;

cout<< " The size is: "<< sizeof(vetor) << endl;

double vetor2[] = {2.1, 3.0, 1.0};

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

cout << "\n"<< vetor2[i] <<endl;

cout<< "The size is: "<<sizeof(vetor2);

return 0;

}

**Ex. 9** Demonstrate automatic counting and aggregate initialization with an array of objects of the class you created in Exercise 3. Add a member function to that class that prints a message. Calculate the size of the array and move through it, calling your new member function.

#include <iostream>

using namespace std;

class Dielectric

{

public: Dielectric(int a);

int Objk1;

void Funkcja1(int Objk1);

~Dielectric();

};

Dielectric::Dielectric(int a)

{

Objk1 = a;

cout<< "\n Constructor invoked! Objk1 = "

<< Objk1 << endl;

}

Dielectric::~Dielectric(){

cout<< "\n Destructor called! Objk1 = " << Objk1<< endl;

}

void Dielectric::Funkcja1(int Objk1)

{

cout <<"\n To jest Funkcja1! a = "<<Objk1;

}

int main()

{

Dielectric fn[] = {Dielectric(1), Dielectric(2), Dielectric(3)};

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

fn[i].Funkcja1(fn[i].Objk1); //

cout<< "\n "<< sizeof(fn)<< endl;

return 0;

}

**7. Function Overloading 7. Function Overloading**

**Ex. 1** Demonstrate automatic counting and aggregate initialization with an array of objects of the class you created in Exercise 3. Add a member function to that class that prints a message. Calculate the size of the array and move through it, calling your new member function.

#include <iostream>

#include <fstream>

#include <string>

#include <cassert>

using namespace std;

class Text{

public:

string Litera;

Text();

Text(string urtu);

string Contenido(string Litera);

~Text();

};

Text::Text(string urtu){

ifstream input;

input.open(urtu);

assert(input);

while (getline(input, Litera))

Litera.c\_str();

}

string Text::Contenido(string Litera){

return Litera;

}

int main()

{

Text Eins;

string ursus;

extern string Litera;

Eins("C:\Proxeiro\_JavaSc");

ursus= Eins.Contenido(Litera);

cout << ursus<< endl;

return 0;

}

**Exercise 2** Create a Message class with a constructor that takes a single string with a default value. Create a private member string, and in the constructor simply assign the argument string to your internal string. Create two overloaded member functions called print( ): one that takes no arguments and simply prints the message stored in the object, and one that takes a string argument, which it prints in addition to the internal message. Does it make sense to use this approach instead of the one used for the constructor?

#include <iostream>

//#include <string>

using namespace std;

class Bericht{

string hs ="& @";

public:

Bericht(string Poet = "suleva!");

~Bericht();

void print();

void print(string);

};

Bericht::Bericht(string Poet){

Poet = "suleva! " + hs;

cout << Poet << endl;

}

Bericht::~Bericht()

{

cout << "Destruction confirme" << endl;

}

void Bericht::print(){

cout <<"\n" << hs << endl;

}

void Bericht::print(string Tao){

cout << Tao <<" "

<< hs << endl;

}

int main(){

Bericht alto;

alto.print();

alto.print("suleva!");

return 0;

}

**Ex. 4** Create a class that contains four member functions, with 0, 1, 2, and 3 int arguments, respectively. Create a main( ) that makes an object of your class and calls each of the member functions. Now modify the class so it has instead a single member function with all the arguments defaulted. Does this change your main( )?

#include <iostream>

using namespace std;

class Klasse{

public:

int Funkci(int a= 0, int = -7, int= -2);

};

int Klasse::Funkci(int a , int er, int x){

int ty= a+ er + x;

return ty;

}

int main()

{

Klasse Rita;

//cout << Rita.Funkci(3,-2,-6);

cout<< Rita.Funkci(3);

cout << endl;

return 0;

}

**Ex.5** Create a function with two arguments and call it from main( ). Now make one of the arguments a “placeholder” (no identifier) and see if your call in main( ) changes.

#include <iostream>

using namespace std;

short Funkcja(short c1, short ){

return (c1);

}

int main()

{

cout<< Funkcja(2,-4);

return 0;

}

Exercises 7, 8, 9, 10 ?

**Ex. 6** Modify Stash3.h and Stash3.cpp to use default arguments in the constructor. Test the constructor by making two different versions of a Stash object.

Stash3.h

Class Stash{

... public:

// Stash(int size); This declaration will be removed.

Stash(int size, int initQuantity = 0);

...};

Stash3.cpp

/\* The following def. Will be removed and the next will be retained only.

Stash::Stash(int sz) {

size = sz;

quantity = 0;

next = 0;

storage = 0;

} \*/

Stash::Stash(int sz, int initQuantity) {

size = sz;

quantity = 0;

next = 0;

storage = 0;

inflate(initQuantity);

}

**Ex. 11** In class Mem, add a bool moved( ) member function that takes the result of a call to pointer( ) and tells you whether the pointer has moved (due to reallocation). Write a main( ) that tests your moved( ) member function. Does it make more sense to use something like moved( ) or to simply call pointer( ) every time you need to access the memory in Mem?

Class Mem{

byte\* mem;

int size;

void ensureMinSize(int minSize);

bool moved(int minsize, int size); //declared before public:

...};

bool Mem::moved(int minsize){

bool call;

if (size < minsize)

call = true;

else call = false;

return call;

}

**8. Constants 8. Constants 8. Constants**

**Ex. 1** Create three const int values, then add them together to produce a value that determines the size of an array in an array definition. Try to compile the same code in C and see what happens (you can generally force your C++ compiler to run as a C compiler by using a commandline flag).

**Ex. 2**. Prove to yourself that the C and C++ compilers really do treat constants differently. Create a global const and use it in a global constantexpression; then compile it under both C and C++.

**Ex. 3**. Create example const definitions for all the built-in types and their variants. Use these in expressions with other consts to make new const definitions. Make sure they compile successfully.

const int fue = -12;

int main() {

const int a =7; const int b = 9; const int c = 1;

int k = a +b+c;

char ar[k+fue];

const unsigned char chun='#';

const float fs = -1.01; const signed short int is = -5;

const long double del = 23; const double tr = del + a;

const float fi = fs -is; const string sutu= "pl" ;

const string sn = sutu +" 8 \* & &";

Ex. 4

Ex.5 Requires <ctime> class functions-objects

Ex. 6

Ex. 7

**Ex. 8** Write two pointers to const long using both forms of the declaration. Point one of them to an array of long. Demonstrate that you can increment or decrement the pointer, but you can’t change what it points to.

int main(){ const long\* lonp1;

long const\* lonp2;

long lonar[4]={ -1, -2, 0, -7};

lonp1=lonar;

lonp2 = lonar;

lonp2 +=3; cout<<\*lonp2;

\*lonp2=-8; //error: assignment of read-only location ‘\* lonp2’

**Ex.9** Write a const pointer to a double, and point it at an array of double. Show that you can change what the pointer points to, but you can’t increment or decrement the pointer.

double roa[3];

double\* const lonp= roa;

--lonp; //error: decrement of read-only variable ‘lonp’

**Ex. 10** Write a const pointer to a const object. Show that you can only read the value that the pointer points to, but you can’t change the pointer or

what it points to.

const double roa[1]={2};

double const\* const lonp= roa;

const double\* const lina= roa;

--lonp; //error: decrement of read-only variable ‘lonp’

**Ex.11** Remove the comment on the error-generating line of code in PointerAssignment.cpp to see the error that your compiler generates.

int main() {

const int e = 2;

int\* v = &e; // Illegal -- e const //error: invalid conversion from ‘const int\*’ to ‘int\*’

**Ex. 12** Create a character array literal with a pointer that points to the beginning of the array. Now use the pointer to modify elements in the array.

char\* Sia = "alto" ; // ISO C++ forbids converting a string constant to ‘char\*’

**Ex. 13** Create a function that takes an argument by value as a const; then try to change that argument in the function body.

void fina(const float shon) {

shon =-2.3;

cout << shon ; } //error: assignment of read-only parameter ‘shon’

**Ex.14** Create a function that takes a float by value. Inside the function, bind a const float& to the argument, and only use the reference from then

on to ensure that the argument is not changed.

const float& fina(float shon) {

const float& Ana = shon;

cout << Ana ;

Ana--; } // error: decrement of read-only reference ‘Ana’

**Ex. 15** Modify ConstReturnValues.cpp removing comments on the error-causing lines one at a time, to see what error messages your compiler generates.

f7(f5()); // error: invalid initialization of non-const reference of type ‘X&’ from an rvalue of type ‘X’

// Causes compile-time errors:

f6() = X(1); //error: passing ‘const X’ as ‘this’ argument discards qualifiers

f6().modify(); // error: passing ‘const X’ as ‘this’ argument discards qualifiers

f7(f6()); //error: invalid initialization of non-const reference of type ‘X&’ from an rvalue of type ‘const //X’

**Ex. 16** Modify ConstPointer.cpp removing comments on the error-causing lines one at a time, to see what error messages your compiler

generates.

char\* cp = v(); // error: invalid conversion from ‘const char\*’ to ‘char\*’

\*w() = 1; //error: assignment of read-only location ‘\* w()’

**Ex. 17** Make a new version of ConstPointer.cpp called ConstReference.cpp which demonstrates references instead of pointers (you may need to look forward to Chapter 11).

void u(const int& cip) {

int i = cip; }

const char\* v() { }

const int& w() {

static int i;

return i; }

int main() { int x = 0; int& ip = x;

const int& cip = x;

const char\* ccp = v();

const int& ccip = w(); // OK

const int& cip2 = w();

**Ex. 18** Make a new version of ConstPointer.cpp called ConstReference.cpp which demonstrates references instead of pointers (you may need to look forward to Chapter 11).

X f() { return X(); }

void g1(X&) {}

int main() {... g1(f()); //error: invalid initialization of non-const

reference of type ‘X&’ from an rvalue of type ‘X’

**Ex. 19** Create a class containing both a const and a non-const float. Initialize these using the constructor initializer list.

class X {

float foa; const float cfo;

public: X(float fio): foa(fio), cfo(fio){cout << foa << "\n cfo: "<< cfo;}

**Ex. 21** Create a class containing a const member that you initialize in the constructor initializer list and an untagged enumeration that you use to determine an array size.

class Alfa { const string suno;

enum {co=3 };

public: float fe[co];

Alfa(string w):suno(w){} };

**Ex. 22** In ConstMember.cpp, remove the const specifier on the member function definition, but leave it on the declaration, to see what kind of compiler error message you get.

class X {...

int f() const; error: candidate is: int X::f() const

int X::f() { return i; } // error: prototype for ‘int X::f()’ does not match any in class ‘X’

**Ex. 23** Create a class with both const and non-const member functions. Create const and non-const objects of this class, and try calling the different types of member functions for the different types of objects.

class X {

public:

int f1() {return 1; }

int f2() const { return -2; }

int main() {const X xi;

xi.f2();

xi.f1(); //Generates error. Constant object calls non const. member function.

**Ex. 24** Create a class with both const and non-const member functions. Try to call a non-const member function from a const member function to see what kind of compiler error message you get.

class X { public:

int f1() {return 1; }

int f2() const { return f1(); } }; 🡨//error: passing ‘const X’ as

‘this’ argument discards qualifiers

Ex. 25 void Z::f() const {

i++; // Error -- const member function

j++; // OK: mutable

} //error: increment of member ‘Z::i’ in read-only object

Casting away constness (const\_cast<Z\*>(this))->i=-23;

**9. Inline Functions 9. Inline Functions 9. Inline Functions**

**Ex.1** Write a program that uses the F( ) macro shown at the beginning of the chapter and demonstrates that it does not expand properly, as described in the text. Repair the macro and show that it works correctly.

#define F (x) (x+1) // With the gap between F and ( it does not compile.)

//Error x cannot be used as function.

#define F(x) (x+1) // Working.

**Ex. 2** Write a program that uses the FLOOR( ) macro shown at the beginning of the chapter. Show the conditions under which it does not work properly.

#define FLOOR(x,b) x>=b?0:1

int main() {

bool k;

int a =0x0f;

k= FLOOR(a bitand 0x0f, 0x07);

cout << k ; //Console for k shows 0.

**Ex. 3** Modify MacroSideEffects.cpp so that BAND( ) works properly.

In main instead of BAND(++a) use:

out << "BAND(++a)=" << **BAND(a+1)** << endl;

**Ex. 4-Ex. 5, Ex. 6** Create two identical functions, f1( ) and f2( ). Inline f1( ) and leave f2( ) as an non-inline function. Use the Standard C Library function clock( ) that is found in <ctime> to mark the starting point and ending points and compare the two functions to see which one is faster. You may need to make repeated calls to the functions inside your timing loop in order to get useful numbers. Experiment with the size and complexity of the code inside the functions in Exercise 4 to see if you can find a break-even point where the inline function and the non-inline function take the same amount of time. If you have them available, try this with different compilers and

note the differences. Prove that inline functions default to internal linkage.

long f1(){

for (int j=0; j< 800; j++)

for (int k=0; k< 800; k++) {

for (int z=0; z< 800; z++)

long sum =j\*k \*z;

}

}

inline long f2(){

for (int j=0; j< 800; j++)

for (int k=0; k< 800; k++) {

for (int z=0; z< 800; z++)

long sum =j\*k \*z;

} }

void print(){

double m = (clock()/CLOCKS\_PER\_SEC);

cout << "\n" << m; }

int main() {

f1();

//f2();

print();

**Ex. 7** Create a class that contains an array of char. Add an inline constructor that uses the Standard C library function memset( ) to initialize the array to the constructor argument (default this to ‘ ’), and an inline member function called print( ) to print out all the characters in the array.

class Renatte{

char c[5];

public:

Renatte(char su =' '){

memset(c, su, 5) ; }

void print(){

for(int i =0; i<5; i++)

cout << c[i] << endl; } or alternatively without memset():

struct Renatte{

char c[5];

public:

Renatte(char su){

for (int u=0; u<5; u++)

c[u] = su; }

void print(){

for(int i =0; i<5; i++)

cout << c[i] << endl; }

};

int main() {

Renatte kobi('!');

kobi.print();

**Ex. 8** Take the NestFriend.cpp example from Chapter 5 and replace all the member functions with inlines. Make them non-in situ inline functions. Also change the initialize( ) functions to constructors.

The lines that changed plus the added keyword inline in front of each function definition.

...public:

Holder();

public:

Pointer(Holder\* h);

inline Holder::Holder() {

memset(a, 0, sz \* sizeof(int)); }...

Holder::Pointer::Pointer(Holder\* rv):h(rv),p(rv->a) { }...

int main() {

Holder h;

Holder::Pointer hp(&h), hp2(&h);

**Ex. 10** Create an enum called Hue containing red, blue, and yellow. Now create a class called Color containing a data member of type Hue and a constructor that sets the Hue from its argument. Add access functions to “get” and “set” the Hue. Make all of the functions inlines.

enum Hue{

rot= 3, blau, gelb

};

class Farbe{

Hue ha;

public:

Farbe(Hue um):ha(um){}

void holHue() {

cout<< ha << endl; }

void stellHue(Hue y){

ha = y; }

};

**Ex. 11** Modify Exercise 10 to use the “accessor” and “mutator” approach.

enum Hue{

rot= 3, blau, gelb };

struct Farbe{

Hue ha;

public:

Farbe(Hue um):ha(um){}

Hue HolHue() const { //Accessor

return ha;} // cout<< ha << endl;

void stellHue(Hue y) { //Mutator

ha = y; }

};

**Ex. 13** Create a class with two inline member functions, such that the first function that’s defined in the class calls the second function, without the need for a forward declaration. Write a main that creates an object of the class and calls the first function.

class Farbe{

int ku;

public:

void feins() {

fzwei(); }

void fzwei(){

cout << ku << endl; }

};

**Ex. 14** Create a class A with an inline default constructor that announces itself. Now make a new class B and put an object of A as a member of B, and give B an inline constructor. Create an array of B objects and see what happens.

class Alpha{

public:

Alpha() {

cout<< "Alpha"<< endl; } };

class Bravo{ Alpha a4;

public:

Bravo(){ cout<< "Bravo" <<endl; } };

**Ex. 17** Correct the TRACE( ) macro as specified in this chapter, and prove that it works correctly.

#define TRACE(s) cerr << #s << endl, s

void f(int g){

cout << g << endl;

//return (g-1); }

int main() {

for (int i =0; i<21; i++){

TRACE(f(i));

**Ex. 18** Modify the FIELD( ) macro so that it also contains an index number. Create a class whose members are composed of calls to the FIELD( ) macro. Add a member function that allows you to look up a field using its index number. Write a main( ) to test the class.

#define FIELD(a) char\* a##\_string; int a##\_size; int a##\_index

class Rekor{

FIELD(uno);

FIELD(dos);

FIELD(tres);

public:

Rekor():uno\_index(1), dos\_index(2)

, tres\_index(3), uno\_string("hello"), dos\_string("allo"),

tres\_string("woohoo"), uno\_size(3){}

void f(int index);

// cout<< tres\_index << endl; };

void Rekor::f(int index){

switch(index){

case 1: cout<< uno\_string << "\t "

<< uno\_size << endl; break;

case 2: cout<< dos\_string << "\t "

<< dos\_size << endl; break;

case 3: cout<< tres\_string << "\t "

<< tres\_size << endl; break; }

}

**Ex. 20** Write a program that takes two command-line arguments: the first is an int and the second is a file name. Use require.h to ensure that you have the right number of arguments, that the int is between 5 and 10, and that the file can successfully be opened.

#include <fstream>

int main(int argc, char\* argv[] )

int i=7;

ifstream in(argv[1]);

assure(in, „file name”);

requireArgs(argc, 2);

require(i>= 5 && i<= 10);

**Ex. 21** Write a program that uses the IFOPEN( ) macro to open a file as an input stream. Note the creation of the ifstream object and its scope.

#include <fstream>

#define IFOPEN(VAR, NAME)\

Ifstream VAR(NAME);\

assure(VAR, NAME);

int main(int argc, char\* argv[] ) {

IFOPEN(in, argv[0]);

**10. Name Control 10. Name Control 10. Name Control**

**Ex. 1** Create a function with a static variable that is a pointer (with a default argument of zero). When the caller provides a value for this argument it is used to point at the beginning of an array of int. If you call the function with a zero argument (using the default argument), the function returns the next value in the array, until it sees a “-1” value in the array (to act as an end-of-array indicator). Exercise this function in main( ).

void fnz(int y[], int );

int main(){

int fo[5];

for (int q = 0; q < 5; q++){

fo[q] = 3\*q;

}

cout << &fo<< endl;

fnz(fo,0);

fnz(fo,5);

fnz(fo,5);

fnz(fo,5);

fnz(fo,0);

fnz(fo,5);

fnz(fo,5);

}

void fnz(int y[], int i= 0 ){

static int\* ew =y;

static int r= 0;

if (i!=0 && (i-r-1)>-1){

cout << ew[r] << endl;

r++;}

else if(i ==0)

cout << ew << endl;

}

**Ex. 2** Create a function that returns the next value in a Fibonacci sequence every time you call it. Add an argument that is a bool with a default value of false such that when you give the argument with true it “resets” the function to the beginning of the Fibonacci sequence. Exercise this function in main( ).

#include <iostream>

using namespace std;

union Ria{

int y=0 ;

int fnz( bool);

void print();

};

int Ria::fnz( bool ek = false){

//static int y = 0;

if(ek == true) {

int y = 0;

return y; }

else {

y++;

return y;

} // cout << ek << endl;

}

void Ria::print(){

cout << y << endl;

}

int main()

{

Ria v;

v.fnz();

v.fnz();

cout << v.fnz() << endl;

}

**Ex. 4** Create a class called Monitor that keeps track of the number of times that its incident( ) member function has been called. Add a print( ) member function that displays the number of incidents. Now create a global function (not a member function) containing a static Monitor object. Each time you call the function it should call incident( ), then print( ) member function to display the incident count. Exercise the function in main( ).

 class Monitor{

    int k;

    public:

    Monitor(int a=0):k(a){}

    void incident();

    void print();

};

void Monitor::incident(){

    ++k;

}

void Monitor::print(){

    cout<< "No of incidents: " << k <<endl;

}

void una(int y){

     static Monitor Esso;

    for(int w =0; w<y; w++)

    Esso.incident();

    Esso.print();

 }

int main()

{ int t;

    cout<< "Enter no of calls: "<< endl;

    cin>> t;

    una(t);

    /\*

    Monitor alto;

    alto.incident();

    alto.incident();

    alto.print();

    alto.incident();

    alto.incident();

    alto.print();\*/

**Ex. 5, Ex.6** Modify the Monitor class from Exercise 4 so that you can decrement( ) the incident count. Make a class Monitor2 that takes as a constructor argument a pointer to a Monitor1, and which stores that pointer and calls incident( ) and print( ). In the destructor for Monitor2, call decrement( ) and print( ). Now make a static object of Monitor2 inside a function. Inside main( ), experiment with calling the function and not calling the function to see what happens with the destructor of Monitor2. Make a global object of Monitor2 and see what happens.

class Monitor{

int k;

public:

Monitor(int a=0):k(a){}

void incident();

void decrement();

void print();

// friend Monitor2;

friend void Rico();

}ek;

void Monitor::incident(){

++k;

}

void Monitor::decrement(){

k-=19;

}

void Monitor::print() {

cout<< "No of incidents: " << k <<endl;

}

void una(int y){

static Monitor Esso;

for(int w =0; w<y; w++)

Esso.incident();

Esso.print();

}

struct Monitor2{

Monitor\* r ;

Monitor2(Monitor\* r);

~Monitor2();

};

Monitor2::Monitor2(Monitor\* r):r(&ek){

r->incident();

r->print();

}

Monitor2:: ~Monitor2(){

r->decrement();

r->print();

}

void Rico(){

static Monitor2 arx(&ek);

}

Monitor2 kor(&ek); //Global Monitor2 object.

int main(){

Rico();

return 0;

}

**Ex. 7** Create a class with a destructor that prints a message and then calls exit( ). Create a global object of this class and see what happens.

The exit() function is not recognized-no cstdlib support.

**Ex.8** In StaticDestructors.cpp, experiment with the order of constructor and destructor calls by calling f( ) and g( ) inside main( ) in different orders. Does your compiler get it right?

//ofstream out("statdest.out"); // Trace file

class Obj {

char c; // Identifier

public:

Obj(char cc) : c(cc) {

cout << "Obj::Obj() for " << c << endl;

}

~Obj() {

cout << "Obj::~Obj() for " << c << endl;

}

};

Obj a('a'); // Global (static storage)

// Constructor & destructor always called

void f() {

static Obj b('b');

}

void g() {

static Obj c('c');

}

int main() {

cout << "inside main()" << endl;

g();

f(); // Calls static constructor for b

// g() not called

cout << "leaving main()" << endl;

} ///:~

**Ex. 9, 10** ?

**Ex. 11** Create a simple class containing an int, a constructor that initializes the int from its argument, a member function to set the int from its argument, and a print( ) function that prints the int. Put your class in a header file, and include the header file in two cpp files. In one cpp file make an instance of your class, and in the other declare that identifier extern and test it inside main( ). Remember, you’ll have to link the two object files or else the linker won’t find the object.

The two cpp files were not tested!

class Obj {

int c;

public:

Obj(int cc):c(cc){}

void set(int sui);

void print();

};

void Obj::set(int sui){

c= sui;

}

void Obj::print(){

cout << c <<"\n";

}

int main() {

Obj Gh(-9);

Gh.print();

Gh.set(4);

Gh.print();

}

**Ex.12,13** Make the instance of the object in Exercise 11 static and verify that it cannot be found by the linker because of this. 13. Declare a function in a header file. Define the function in one cpp file and call it inside main( ) in a second cpp file. Compile and verify that it works. Now change the function definition so that it is static and verify that the linker cannot find it.

The same reason as ex. 11 due to the online compiler.

**Ex. 14** Modify Volatile.cpp from Chapter 8 to make comm::isr( ) something that could actually work as an interrupt service routine. Hint: an interrupt service routine doesn’t take any arguments.

//: C08:Volatile.cpp

// From Thinking in C++, 2nd Edition

void Comm::isr() volatile {

flag = 0;

for(int index =0; index<100;){ //This loop added and the cout line.

buf[index++] = byte;

cout << buf[index]<< endl;}

// Wrap to beginning of buffer:

if(index >= bufsize) index = 0; }

**Ex. 16** Modify Volatile.cpp from Chapter 8 to make comm::isr( ) something that could actually work as an interrupt service routine. Hint: an interrupt service routine doesn’t take any arguments.

namespace jon{

int f1(int i1){

return (i1-=1); }

int f2(int i2){

return i2-=2; }

}

namespace jon{

int f3(int i3){

return i3-=9; }

void f4(int f1);

}

namespace einfach = jon;

void funcj(){

int m\_= einfach::f2(1);

cout << m\_ <<"\n" ; }

void funcj2() {

using namespace einfach;

cout<< f3(3); }

int main() {

funcj();

funcj2();

return 0;

}

**Ex. 18** Using the header file from Exercise 17, show that the names in an unnamed namespace are automatically available in a translation unit without qualification.

namespace {

int f1(int i1){

return (i1-=1);} }

int main()

{ cout << f1(-5);

**Ex. 19** Modify FriendInjection.cpp to add a definition for the friend function and to call the function inside main( ).

namespace Me {

class Us {

friend void you();

};

void you(){

cout << "you des names"<< endl; }

}

//using namespace Me;

void you(int u){

cout << "Another you outside Me: "<< u; }

int main() {

Me::you();

**Ex. 20** In Arithmetic.cpp, demonstrate that the using directive does not extend outside the function in which the directive was made.

Declaring: int main(){

Integer x; //and the error message is : ‘Integer’ was not declared in this scope

**Ex. 21** Repair the problem in OverridingAmbiguity.cpp, first with scope resolution, then instead with a using declaration that forces the compiler to choose one of the identical function names.

void s() {

//using namespace Math;

using namespace Calculation;

using Math::divide; // divide of Math overrides divide of Calculation

divide(1,2);

}

int main() {

s();

**Ex. 22** In two header files, create two namespaces, each containing a class (with all inline definitions) with a name identical to that in the other namespace. Create a cpp file that includes both header files. Create a function, and inside the function use the using directive to introduce both namespaces. Try creating an object of the class and see what happens. Make the using directives global (outside of the function) to see if it makes any difference. Repair the problem using scope resolution, and create objects of both classes.

namespace Klas{

class tina{

int u;

public:

inline void Juli(int d){u= d;}

inline void print(){cout<< u;}

};

}

namespace tina{

class Klas{

int u2;

public:

inline void tania(int h){u2 = h ;}

inline void print(){cout<< u2;}

}; }

// using namespace Klas;

// using namespace tina;

void funa(){

tina::Klas tery;

tery.tania(-8);

tery.print();

}

int main()

{

funa();

**Ex.23** Repair the problem in Exercise 22 with a using declaration that forces the compiler to choose one of the identical class names.

using tina::Klas ;

Klas tery;

tery.tania(-8);

tery.print(); }

**Ex. 24** Extract the namespace declarations in BobsSuperDuperLibrary.cpp and UnnamedNamespaces.cpp and put them in separate header files, giving the unnamed namespace a name in the process. In a third header file create a new namespace that combines the elements of the other two namespaces with using declarations. In main( ), introduce your new namespace with a using directive and access all the elements of your namespace.

namespace zon{

using Bob::Widget;

using Bob::Poppit;

using ::Arm;

using ::Leg;

using ::Head;

using ::Robot;

using ::i;

using ::j;

using ::k; }

namespace rick = zon;

int main()

{using namespace rick;

Leg hua; hua.hol(7);

hua.print(); cout<<j;

**Ex. 25** Create a header file that includes <string> and <iostream> but does not use any using directives or using declarations. Add “include guards” as you’ve seen in the header files in this book. Create a class with all inline functions that contains a string member, with a constructor that initializes that string from its argument and a print( ) function that displays the string. Create a cpp file and exercise your class in main( ).

Does not recongize string type and cout without std namespace!

**Ex. 26** Create a class containing a static double and long. Write a static member function that prints out the values.

class Widget {

static double qur;

static long ji;

public:

static void print(){

cout << " "<< qur;

}

};

double Widget::qur = -2;

int main()

{

Widget ser;

ser.print();

**Ex. 27** Create a class containing an int, a constructor that initializes the int from its argument, and a print( ) function to display the int. Now create a second class that contains a static object of the first one. Add a static member function that calls the static object’s print( ) function. Exercise your class in main( ).

class Widget {

int ras;

public:

Widget():ras(-9){}

void print(){cout << ras; }

};

class gary {

public:

static Widget lucy;

static void f(){

lucy.print(); }

};

Widget gary::lucy;

int main()

{ //either this:

gary sup;

sup.f();

gary::f(); //Or that:

**Ex.28** Create a class containing both a const and a non-const static array of int. Write static methods to print out the arrays. Exercise your class in main( ).

class gary {

const int ar[3] = {4, -9, 78 };

static const int ars[];

public:

void meto();

static void meto2();

};

const int gary::ars[] = {

-1, -4 };

void gary::meto(){

for(int j= 0; j< 5; j++)

cout << ar[j]<< endl;

cout << sizeof(ar); }

int main() { gary rinho;

rinho.meto();

**Ex. 29** Create a class containing a string, with a constructor that initializes the string from its argument, and a print( ) function to display the string. Create another class that contains both const and non-const static arrays of objects of the first class, and static methods to print out these arrays. Exercise this second class in main( ).

class G{

string ef;

public:

G(string tengo):ef(tengo){}

void print(){

cout<< ef<< endl; }

};

class G2{

static G ulm[];

public:

static void f\_ulm();

// static void f\_rim();

};

G G2::ulm[] = { G("Cheerio"), G("hooray")};

void G2::f\_ulm() {

for (int j=0; j< 2; j++)

ulm[j].print(); }

int main()

{ G2 iso;

iso.f\_ulm();

**Ex. 30** Create a struct that contains an int and a default constructor that initializes the int to zero. Make this struct local to a function. Inside that function, create an array of objects of your struct and demonstrate that each int in the array has automatically been initialized to zero.

void funz(){

struct Widget{

int tempo;

Widget(int roko):tempo(roko){}

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

};

Widget Widge(0), Widet(0);

Widget pam[] = {Widge, Widet};

pam[0].print();

pam[1].print(); }

int main()

{ funz();

**Ex. 31** Create a class that represents a printer connection, and that only allows you to have one printer.

Copy from Singleton.cpp on p. 399. Substitute Egg with

Printer\_Connection and e with Printer which is the unique object.

**Ex. 32** In a header file, create a class Mirror that contains two data members: a pointer to a Mirror object and a bool. Give it two constructors: the default constructor initializes the bool to true and the Mirror pointer to zero. The second constructor takes as an argument a pointer to a Mirror object, which it assigns to the object’s internal pointer; it sets the bool to false. Add a member function test( ): if the object’s pointer is nonzero, it returns the value of test( ) called through the pointer. If the pointer is zero, it returns the bool. Now create five cpp files, each of which includes the Mirror header. The first cpp file defines a global Mirror object using the default constructor. The second file declares the object in the first file as extern, and defines a global Mirror object using the second constructor, with a pointer to the first object. Keep doing this until you reach the last file, which will also contain a global object definition. In that file, main( ) should call the test( ) function and report the result. If the result is true, find out how to change the linking order for your linker and change it until the result is false.

**Ex. 33**. Repair the problem in Exercise 32 using technique one shown in this book.

**Ex. 34**. Repair the problem in Exercise 32 using technique two shown in this book.

Not completed

class Mirror {

Mirror\* pot;

bool ghi;

public:

Mirror():ghi(true), pot(0){}

Mirror(Mirror\* zu):pot(zu), ghi(false){}

bool test(){

if(pot !=0)

return pot->test();

else return pot->ghi; }

};

Mirror Mr1;

extern Mirror Mr1;

Mirror Mr2(&Mr1);

**Factory Method Design Pattern in C++ ?**

struct Service {

virtual void meto1()= 0;

void meto2();

};

struct Una : Service {

Una() {}

void meto1() { cout << "meto1()" << endl; }

void meto2() { cout << "meto2()" << endl; }

};

struct ServiFacto {

virtual Una\* getServi() = 0;

};

struct IMServiFacto : ServiFacto {

Una\* getServi() {

return new Una();

}

};

void serviConsu(IMServiFacto faco) {

Una\* s = faco.getServi();

s->meto1();

}

int main() {

IMServiFacto tula;

serviConsu(tula);

**11. References & the Copy-Constructor 11. References & the Copy-Constructor**

**Ex.3** Write a program in which you try to (1) Create a reference that is not initialized when it is created. (2) Change a reference to refer to another object after it is initialized. (3) Create a NULL reference.

class Kls1{

float h,q;

public:

Kls1(){}

Kls1(float r, float z):h(r), q(z){}

void f1() const {

cout <<h<<"\t"<< q<< endl; } };

int main()

{ Kls1 Mango1, Pear(-1.2, 7.0);

//float& m; compile time error withoug initialization

Kls1& ng = Mango1;

ng = Pear;

ng.f1(); //Prints the contents of Pear

**Ex. 4** Write a function that takes a pointer argument, modifies what the pointer points to, and then returns the destination of the pointer as a reference.

float& f(float\* pf) {

\*pf = -0.082;

float& ref = \*pf;

return ref; }

int main() { float\* flp;

float& x=f(flp);

float g= x;

**Ex. 5** Write a function that takes a pointer argument, modifies what the pointer points to, and then returns the destination of the pointer as a reference.

class Kls1{ public:

void f1() const {

cout <<"f1()"<< endl; }

void f2() const;

void f3(); };

void Kls1::f2() const {

cout << "f2()" << endl; }

void Kls1::f3(){

cout <<"f3()" << endl; }

void fp(const Kls1\* fp1){

fp1->f1(); // Or

fp1->Kls1::f2();

//fp1 -> Kls1::f3(); compile time error }

int main() { //By the use of constant pointer

const Kls1\* fp1;

fp(fp1);

//And by use of reference

void fp(const Kls1& fp1){

const Kls1 fernando;

fernando.f1(); // Or

fernando.Kls1::f2();

//fernando.Kls1::f3(); //compile time error }

int main() {

Kls1 fernando;

fp(fernando);

// p. 411 NULL references are not allowed

**Ex. 6** Take the code fragments at the beginning of the section titled “Pointer references” and turn them into a program.

float\*\* f(float\*\* pf) {

\*\*pf = -1;

return pf; }

int main() {

float\* vk;

f(&vk);

cout << \*vk << endl;

**Ex. 7** Create a function that takes an argument of a reference to a pointer to a pointer and modifies that argument. In main( ), call the function.

int\*\*& f(int\*\*& uj){

\*\*uj =-3;

return uj; }

int main() {

int\*\* q2;

int\*\*& rf=q2;

f(rf);

cout << \*\*rf << endl;

**Ex. 8** Create a function that takes a char& argument and modifies that argument. In main( ), print out a char variable, call your function for that variable, and print it out again to prove to yourself that it has been changed.

char& f(char& m){

m ='@';

return m;}

int main(){

char d= '9';

char& rt =d;

cout<<f(rt);

**Ex. 9** Write a class that has a const member function and a non-const member function. Write three functions that take an object of that class as an argument; the first takes it by value, the second by reference, and the third by const reference. Inside the functions, try to call both member functions of your class and explain the results.

int Venus::fv1()const {

int w =8;

return w; }

int Venus::fv2() {

int h= 3;

return h; }

int fG1(Venus m) {

int F=m.fv1();

int F2= m.fv2();

return F-F2; }

int fG2(Venus& M) {

int F=M.fv1();

int F2=M.fv2();

return F-F2; }

int fG3(const Venus& m) {

int F= m.fv1();

//int F2= m.fv2(); // Compilation error

return F; }

int main() {

Venus Omega;

Venus& Sara = Omega;

const Venus& Mary = Omega;

fG1(Omega);

fG3(Mary);

fG2(Sara);

**Ex. 10** , 11, 12, 13, 14 Need for assembly code production

**Ex. 15** Write code to prove that the compiler automatically synthesizes a copy-constructor if you don’t create one yourself. Prove that the synthesized copy-constructor performs a bitcopy of primitive types and calls the copy-constructor of user-defined types.

Refer to the program „DefaultCopyConstructor.cpp”,p.421-422

**Ex. 16** Write a class with a copy-constructor that announces itself to cout. Now create a function that passes an object of your new class in by value and another one that creates a local object of your new class and returns it by value. Call these functions to prove to yourself that the copyconstructor is indeed quietly called when passing and returning objects by value.

class Klasse1{

static int Kopierung;

public:

Klasse1() {cout<< "Beschaffen \n";}

Klasse1(const Klasse1&){

++Kopierung;

cout << "Klasse& beschafft "

<< Kopierung<<endl; }

~Klasse1() { --Kopierung;

cout << "Zerstoereung: "<< Kopierung<< endl; }

};

int Klasse1::Kopierung = 0;

void f1(Klasse1 O) { cout << "f1"<< endl; }

Klasse1 f2(){

Klasse1 sum;

return sum; }

int main() {

Klasse1 Ojk;

f1(Ojk);

f2();

**Ex.17** Create a class that contains a double\*. The constructor initializes the double\* by calling new double and assigning a value to the resulting storage from the constructor argument. The destructor prints the value that’s pointed to, assigns that value to -1, calls delete for the storage, and then sets the pointer to zero. Now create a function that takes an object of your class by value, and call this function in main( ). What happens? Fix the problem by writing a copy-constructor.

class Klasse1{ **Not functional!**

double\* k;

public:

Klasse1(Klasse1&){

cout << "Kopierung\n"; }

Klasse1(double e):k(new double){

k = &e;

cout << \*k<< endl; }

~Klasse1(){ cout << \*k<< endl;

// \*k = -1; produces segmentation fault

cout << \*k<< endl;

// delete []k;

//\*k = 0; produces segmentation fault

// cout<< \*k; }

};

void f1(Klasse1 Omo) { cout << "f1"<< endl; }

int main() {

Klasse1 Ojk(54);

f1(Ojk);

**Ex. 18** Create a class with a constructor that looks like a copy-constructor, but that has an extra argument with a default value. Show that this is still used as the copy-constructor.

class Klasse1{

public:

Klasse1(Klasse1&, double putra=3){

cout << "Kopierung\n"<< putra; }

Klasse1(){ cout << endl; }

};

void f1(Klasse1 Omo){

Omo; }

int main() {

Klasse1 Ojk;

f1(Ojk);

**Ex. 19** Create a class with a copy-constructor that announces itself. Make a second class containing a member object of the first class, but do not create a copy-constructor. Show that the synthesized copy-constructor in the second class automatically calls the copy-constructor of the first class.

class Klasse1{

public:

Klasse1(){}

Klasse1(Klasse1&){

cout << "Kopierung\n"; }

class Klasse2{

Klasse1 Ojk1;

public:

Klasse2(){} };

int main() {

Klasse2 Ojk2;

Klasse2 Ojk22 = Ojk2;

Ex. 20

class Klasse1{

public:

Klasse1(){}

Klasse1(Klasse1&) { cout << "Kopierung\n"; } };

Klasse1 f1(Klasse1 Ojk){ return Ojk; }

void f2( const Klasse1& ){

cout << "f2"; }

int main() {

Klasse1 Ojk1;

f2(f1(Ojk1));

**Ex. 21** Create a simple class without a copy-constructor, and a simple function that takes an object of that class by value. Now change your class by adding a private declaration (only) for the copy-constructor.

class Klasse1{

Klasse1(Klasse1&){

cout << "Kopierung\n"; }

public:

Klasse1(){} };

Klasse1 f1(Klasse1 Ojk){

cout << "f1\n";

return Ojk; }

int main() {

Klasse1 Ojk1;

f1(Ojk1); // Compiler error: Klasse1::Klasse1(Klasse1&) is private.

**Ex. 22** This exercise creates an alternative to using the copy-constructor. Create a class X and declare (but don’t define) a private copy-constructor. Make a public clone( ) function as a const member function that returns a copy of the object that is created using new. Now write a function that takes as an argument a const X& and clones a local copy that can be modified. The drawback to this approach is that you are responsible for explicitly destroying the cloned object (using delete) when you’re done with it.

**//Not functional!**

class Klasse1{

Klasse1(Klasse1&);

public:

Klasse1(){}

Klasse1\* clone() const;

};

Klasse1\* Klasse1::clone() const{

Klasse1\* Ojk = new Klasse1();

return Ojk; }

void f1(const Klasse1& Olaf){

Klasse1\* Jurgen;

Klasse1 Franz;

Jurgen=Olaf.clone();

delete Jurgen; }

int main() {

Klasse1 Ojk1;

f1(Ojk1);

**Ex. 24** Create a class containing a double and a print( ) function that prints the double. In main( ), create pointers to members for both the data member and the function in your class. Create an object of your class and a pointer to that object, and manipulate both class elements via your pointers to members, using both the object and the pointer to the object.

class Klasse {

public:

double ur;

void print(){

cout << ur<<"\n"; }

};

void (Klasse::\*fpin)()=&Klasse::print;

int main(){

Klasse Angus;

Klasse\* sp = &Angus;

double Klasse::\*pintodu =&Klasse::ur;

sp->\*pintodu =-9; //Object pointer calls member pointer

(sp->\*fpin)(); //Object pointer calls function pointer

sp->print();

Angus.\*pintodu = -2; //Object calls member pointer

Angus.print(); //Object calls member function

(Angus.\*fpin)(); //Object calls member function pointer

**Ex. 25** Create a class containing an array of int. Can you index through this array using a pointer to member?

**Non** **functional !**

class Klasse {

public:

// const int sz =7;

double\* em;

double ur[];

void print(){

cout << "\n"; } };

int main(){

Klasse Angus;

Klasse\* sp;

double\* Klasse::\*pintodu =&Klasse::em;

sp->\*pintodu =&Klasse::ur;

int a[3];

int\* pe = a;

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

pe[i] =i;

cout <<pe[i]<< endl; }

**Ex. 26** Modify PmemFunDefinition.cpp by adding an overloaded member function f( ) (you can determine the argument list that causes the overload). Now make a second pointer to member, assign it to the overloaded version of f( ), and call the function through that pointer. How does the overload resolution happen in this case?

class A{

public:

int f(float) {return 1;}

int f(char) {return -1;} };

int (A::\*fpov)(float) =&A::f;

int (A::\*fpv)(char) =&A::f;

int main() { A\* op;

cout<<(op->\*fpv)(-2.2);

**Ex. 27, 28** Start with FunctionTable.cpp from Chapter 3. Create a class that contains a vector of pointers to functions, with add( ) and remove( ) member functions to add and remove pointers to functions. Add a run( ) function that moves through the vector and calls all of the functions. Modify exercise 27 so that it works with pointers to member functions instead.

**Non Functional!**

class Alfa{ public:

void (\*fp)(double);

//vector <(\*fp)> V\_p;

void Add();

void Remove();

void Run(); };

void Alfa::Add(){

// void (\*fp)(double) = &Bravo::f2;

//Should call the push\_back() for vector

elements }

void Alfa::Remove(){

/\*Should call the erase function\*/ }

void f1(double g){

cout<< g<< endl; }

void f2(double g){

cout<< g<< endl; }

void f3(double g){

cout<< g<< endl; }

int main() {Alfa Ojk;

void (Alfa::\*fp)(double) =&Alfa::f2;

(Ojk.\*fp)(4);

**12. Operator Overloading 12. Operator Overloading**

**Ex. 1** Create a simple class with an overloaded operator++. Try calling this operator in both pre- and postfix form and see what kind of compiler warning you get.

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** Create a simple class containing an int and overload the operator+ as a member function. Also provide a print( ) member function that takes an ostream& as an argument and prints to that ostream&. Test your class to show that it works correctly. Add a binary operator- to Exercise 2 as a member function. Demonstrate that you can use your objects in complex expressions like a + b – c. Add an operator++ and operator-- to Exercise 2, both the prefix and the postfix versions, such that they return the incremented or decremented object. Make sure that the postfix versions return the correct value.

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** Modify the increment and decrement operators in Exercise 4 so that the prefix versions are return a non-const reference and the postfix versions are return a const object. Show that they work correctly and explain why this would be done in practice.

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** Change the print( ) function in Exercise 2 so that it is the overloaded operator<< as in ostreamOperatorOverloading.cpp.

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

cout << c.c << endl;

return cout; }

int main() {

Alfa ora; cout << ora;

**Ex. 7** Modify Exercise 3 so that the operator+ and operator- are non-member functions. Demonstrate that they still work correctly.

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** Add the unary operator- to Exercise 2 and demonstrate that it works correctly.

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** Create a class that contains a single private char. Overload the iostream operators << and >> (as in

IostreamOperatorOverloading.cpp) and test them. You can test them with fstreams, stringstreams, and cin and cout.

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** Determine the dummy constant value that your compiler passes for postfix operator++ and operator--.

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** Determine the dummy constant value that your compiler passes for postfix operator++ and operator--.

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** Modify Exercise 11 so that the return value optimization is used, if you have not already done so. **Return value optimization ?**

**Ex. 13** Create a class that contains a pointer, and demonstrate that if you allow the compiler to synthesize the operator= the result of using that operator will be pointers that are aliased to the same storage. Now fix the problem by defining your own operator= and demonstrate that it corrects the aliasing. Make sure you check for self-assignment and handle that case properly.

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** Write a class called Bird that contains a string member and a static int. In the default constructor, use the int to automatically generate an identifier that you build in the string, along with the name of the class (Bird #1, Bird #2, etc.). Add an operator<< for ostreams to print out the Bird objects. Write an assignment operator= and a copy-constructor. In main( ), verify that everything works correctly.

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** Write a class called BirdHouse that contains an object, a pointer and a reference for class Bird from Exercise 14. The constructor should take the three Birds as arguments. Add an operator<< for ostreams for BirdHouse. Write anDisallow the assignment operator= and a copy-constructor. In main( ), verify that everything works correctly.

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** Add an int data member to both Bird and BirdHouse in Exercise 15. Add member operators +, -, \*, and / that use the int members to perform the operations on the respective members. Verify that these work.

//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.** **18** Add an operator-- to SmartPointer.cpp and NestedSmartPointer.cpp.

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** Modify CopyingVsInitialization.cpp so that all of the constructors print a message that tells you what’s going on. Now verify that the two forms of calls to the copy-constructor (the assignment form and the parenthesized form) are equivalent.

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** Attempt to create a non-member operator= for a class and see what kind of compiler message you get.

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** Create a class with an copy-constructor assignment operator that has a second argument, a string that has a default value that says “CC op= call.” Create a function that takes assigns an object of your class by value to another one and show that your copy-constructorassignment operator is called correctly.

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** In CopyingWithPointers.cpp, remove the operator= in DogHouse and show that the compiler-synthesized operator= correctly copies the string but simply aliases the Dog pointer.

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** Create a class containing a string as a data member. Initialize the string in the constructor, but do not create a copy-constructor or operator=. Make a second class that has a member object of your first class; do not create a copy-constructor or operator= for this class either. Demonstrate that the copy-constructor and operator= are properly synthesized by the compiler.

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** Add an operator->\* to NestedSmartPointer.cpp.

Nonfunctional!

#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** Create two classes, Apple and Orange. In Apple, create a constructor that takes an Orange as an argument. Create a function that takes an Apple and call that function with an Orange to show that it works. Now make the Apple constructor explicit to demonstrate that the automatic type conversion is thus prevented. Modify the call to your function so that the conversion is made explicitly and thus succeeds.

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** Add a global operator\* to ReflexivityInOverloading.cpp and demonstrate that it is reflexive.

// 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** Create two classes and create an operator+ and the conversion functions such that addition is reflexive for the two classes.

**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 13. Dynamic Object Creation**

**Ex. 1** Create a class Counted that contains an int id and a static int count. The default constructor should begin:

Counted( ) : id(count++) {. It should also print its id and that it’s being created. The destructor should print that it’s being destroyed and its id. Test your class.

**Ex.2** Prove to yourself that new and delete always call the constructors and destructors by creating an object of class Counted (from Exercise 1) with new and destroying it with delete. Also create and destroy an array of these objects on the heap.

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** Create a PStash object and fill it with new objects from Exercise 1. Observe what happens when this PStash object goes out of scope and its destructor is called.

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. 6** Repeat Exercise 5 using a PStash.

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. 8** Dynamically create an array of objects of class Counted (from Exercise 1). Call delete for the resulting pointer, without the square brackets.

int main() {

Gezaehlt\* Gzp = new Gezaehlt[2];

delete Gzp;

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

**Ex. 9** Create an object of class Counted (from Exercise 1) using new, cast the resulting pointer to a void\*, and delete that.

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. 11** Create a class with an overloaded operator new and delete, both the single-object versions and the array versions. Demonstrate that both versions work.

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. 13** Modify NoMemory.cpp so that it contains an array of int and so that it actually allocates memory instead of throwing bad\_alloc. In main( ), set up a while loop like the one in NewHandler.cpp to run out of memory and see what happens if your operator new does not test to see if the memory is successfully allocated. Then add the check to your operator new and throw bad\_alloc.

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** Create a class with a placement new with a second argument of type string. The class should contain a static vector<string> where the second new argument is stored. The placement new should allocate storage as normal. In main( ), make calls to your placement new with string arguments that describe the calls (you may want to use the preprocessor’s \_\_FILE\_\_ and \_\_LINE\_\_ macros).

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** Modify ArrayOperatorNew.cpp by adding a static vector<Widget\*> that adds each Widget address that is allocated in operator new( ) and removes it when it is released via operator delete( ). (You may need to look up information about vector in your Standard C++ Library documentation or in the 2nd volume of this book, available at the Web site.) Create a second class called MemoryChecker that has a destructor that prints out the number of Widget pointers in your vector. Create a program with a single global instance of MemoryChecker and in main( ), dynamically allocate and destroy several objects and arrays of Widget. Show that MemoryChecker reveals memory leaks.

**Incomplete!**

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 14. Inheritance – Composition**

**Ex. 1** Modify Car.cpp so that it also inherits from a class called Vehicle, placing appropriate member functions in Vehicle (that is, make up some member functions). Add a nondefault constructor to Vehicle, which you must call inside Car’s constructor.

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** Create two classes, A and B, with default constructors that announce themselves. Inherit a new class called C from A, and create a member object of B in C, but do not create a constructor for C. Create an object of class C and observe the results.

class A {

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

class B {

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

class C : public A{ B beta; };

**Ex. 3** Create a three-level hierarchy of classes with default constructors, along with destructors, both of which announce themselves to cout. Verify that for an object of the most derived type, all three constructors and destructors are automatically called. Explain the order in which the calls are made.

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** Modify Combined.cpp to add another level of inheritance and a new member object. Add code to show when the constructors and destructors are being called.

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** In Combined.cpp, create a class D that inherits from B and has a member object of class C. Add code to show when the constructors and destructors are being called.

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** Modify Order.cpp to add another level of inheritance Derived3 with member objects of class Member4 and Member5. Trace the output of the program.

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** In NameHiding.cpp, verify that in Derived2, Derived3, and Derived4, none of the base-class versions of f( ) are available.

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** Modify NameHiding.cpp by adding three overloaded functions named h( ) to Base, and show that redefining one of them in a derived class hides the others.

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. 10** Write a class containing a long and use the psuedo-constructor call syntax in the constructor to initialize the long.

class Base {long lona;

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

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

**Ex. 11** Create a class called Asteroid. Use inheritance to specialize the PStash class in Chapter 13 (PStash.h & PStash.cpp) so that it accepts and returns Asteroid pointers. Also modify PStashTest.cpp to test your classes. Change the class so PStash is a member object.

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. 13** In SynthesizedFunctions.cpp, modify Chess to give it a default constructor, copy-constructor, and assignment operator. Demonstrate that you’ve written these correctly.

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** Create two classes called Traveler and Pager without default constructors, but with constructors that take an argument of type string, which they simply copy to an internal string variable. For each class, write the correct copy-constructor and assignment operator. Now inherit a class BusinessTraveler from Traveler and give it a member object of type Pager. Write the correct default constructor, a constructor that takes a string argument, a copy-constructor, and an assignment operator.

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** Create a class with two static member functions. Inherit from this class and redefine one of the member functions. Show that the other is hidden in the derived class.

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. 17** Use private and protected inheritance to create two new classes from a base class. Then attempt to upcast objects of the derived class to the base class.

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** In Protected.cpp, add a member function in Derived that calls the protected Base member read( ).

class Derived : public Base {

public:

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

**Ex. 19** Change Protected.cpp so that Derived is using protected inheritance. See if you can call value( ) for a Derived object.

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

**Ex. 20** Create a class called SpaceShip with a fly( ) method. Inherit Shuttle from SpaceShip and add a land( ) method. Create a new Shuttle, upcast by pointer or reference to a SpaceShip, and try to call the land( ) method.

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** Modify Instrument.cpp to add a prepare( ) method to Instrument. Call prepare( ) inside tune( ).

class Instrument { public:

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

void tune(Instrument& i) {

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

i.vorbereiten(Cflat);

**Ex. 22** Modify Instrument.cpp so that play( ) prints a message to cout, and Wind redefines play( ) to print a different message to cout. Run the program and explain why you probably wouldn’t want this behavior. Now put the virtual keyword (which you will learn about in Chapter 15) in front of the play( ) declaration in Instrument and observe the change in the behavior.

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** In CopyConstructor.cpp, inherit a new class from Child and give it a Memberm. Write a proper constructor, copy-constructor, operator=, and operator<< for ostreams, and test the class in main( ).

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** Take the example CopyConstructor.cpp and modify it by adding your own copy-constructor to Child without calling the base-class copyconstructor and see what happens. Fix the problem by making a proper explicit call to the base-class copy constructor in the constructorinitializer list of the Child copy-constructor.

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. 26** Create a class Rock with a default constructor, a copy-constructor, an assignment operator, and a destructor, all of which announce to cout that they’ve been called. In main( ), create a vector<Rock> (that is, hold Rock objects by value) and add some Rocks. Run the program and explain the output you get. Note whether the destructors are called for the Rock objects in the vector. Now repeat the exercise with a vector<Rock\*>. Is it possible to create a vector<Rock&>?

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** This exercise creates the design pattern called proxy. Start with a base class Subject and give it three functions: f( ), g( ), and h( ). Now inherit a class Proxy and two classes Implementation1 and Implementation2 from Subject. Proxy should contain a pointer to a Subject, and all the member functions for Proxy should just turn around and make the same calls through the Subject pointer. The Proxy constructor takes a pointer to a Subject that is installed in the Proxy (usually by the constructor). In main( ), create two different Proxy objects that use the two different implementations. Now modify Proxy so that you can dynamically change implementations.

**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** Modify ArrayOperatorNew.cpp from Chapter 13 to show that, if you inherit from Widget, the allocation still works correctly. Explain why inheritance in Framis.cpp from Chapter 13 would not work correctly.

class Alfa:public Widget{

public: };

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

Alfa\* wa = new Alfa[25];

**Ex. 29** Modify Framis.cpp from Chapter 13 by inheriting from Framis and creating new versions of new and delete for your derived class. Demonstrate that they work correctly.

**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 15: Polymorphism 15: Polymorphism**

**Ex. 1** Create a simple “shape” hierarchy: a base class called Shape and derived classes called Circle, Square, and Triangle. In the base class, make a virtual function called draw( ), and override this in the derived classes. Make an array of pointers to Shape objects that you create on the heap (and thus perform upcasting of the pointers), and call draw( ) through the base-class pointers, to verify the behavior of the virtual function. If your debugger supports it, single-step through the code.

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** Modify Exercise 1 so draw( ) is a pure virtual function. Try creating an object of type Shape. Try to call the pure virtual function inside the constructor and see what happens. Leaving it as a pure virtual, give draw( ) a definition.

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** Expanding on Exercise 2, create a function that takes a Shape object by value and try to upcast a derived object in as an argument. See what happens. Fix the function by taking a reference to the Shape object.

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** Modify C14:Combined.cpp so that f( ) is virtual in the base class. Change main( ) to perform an upcast and a virtual call.

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** Modify Instrument3.cpp by adding a virtual prepare( ) function. Call prepare( ) inside tune( ).

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** Create an inheritance hierarchy of Rodent: Mouse, Gerbil, Hamster, etc. In the base class, provide methods that are common to all Rodents, and redefine these in the derived classes to perform different behaviors depending on the specific type of Rodent. Create an array of pointers to Rodent, fill it with different specific types of Rodents, and call your base-class methods to see what happens.

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** Modify Exercise 6 so that you use a vector<Rodent\*> instead of an array of pointers. Make sure that memory is cleaned up properly.

**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** Starting with the previous Rodent hierarchy, inherit BlueHamster from Hamster (yes, there is such a thing; I had one when I was a kid), override the base-class methods, and show that the code that calls the base-class methods doesn’t need to change in order to accommodate the new type.

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** Starting with the previous Rodent hierarchy, add a non virtual destructor, create an object of class Hamster using new, upcast the pointer to a Rodent\*, and delete the pointer to show that it doesn’t call all the destructors in the hierarchy. Change the destructor to be virtual and demonstrate that the behavior is now correct.

//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** Starting with the previous Rodent hierarchy, modify Rodent so it is a pure abstract base class.

class Rodent { public:

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

virtual void fernando() const = 0; };

**Ex. 11** Create an air-traffic control system with base-class Aircraft and various derived types. Create a Tower class with a vector<Aircraft\*> that sends the appropriate messages to the various aircraft under its control.

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

Must be able first to do ex. 7.

**Ex.13** In Early.cpp, make Pet a pure abstract base class.

virtual string speak()=0;

**Ex. 14** In AddingVirtuals.cpp, make all the member functions of Pet pure virtuals, but provide a definition for name( ). Fix Dog as necessary, using the base-class definition of name( ).

class Pet {

virtual string name() const=0;

virtual string speak() const=0;

**Ex. 15** Write a small program to show the difference between calling a virtual function inside a normal member function and calling a virtual function inside a constructor. The program should prove that the two calls produce different results.

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** Modify VirtualsInDestructors.cpp by inheriting a class from Derived and overriding f( ) and the destructor. In main( ), create and upcast an object of your new type, then delete it.

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** Take Exercise 16 and add calls to f( ) in each destructor. Explain what happens.

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** Create a class that has a data member and a derived class that adds another data member. Write a non-member function that takes an object of the base class by value and prints out the size of that object using sizeof. In main( ) create an object of the derived class, print out its size, and then call your function. Explain what happens.

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** Write a class with one virtual function and one non-virtual function. Inherit a new class, make an object of this class, and upcast to a pointer of the base-class type. Use the clock( ) function found in <ctime> (you’ll need to look this up in your local C library guide) to measure the difference between a virtual call and non-virtual call. You’ll need to make multiple calls to each function inside your timing loop in order to see the difference.

**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** Modify C14:Order.cpp by adding a virtual function in the base class of the CLASS macro (have it print something) and by making the destructor virtual. Make objects of the various subclasses and upcast them to the base class. Verify that the virtual behavior works and that proper construction and destruction takes place.

#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** Write a class with three overloaded virtual functions. Inherit a new class from this and override one of the functions. Create an object of your derived class. Can you call all the base class functions through the derived-class object? Upcast the address of the object to the base. Can you call all three functions through the base? Remove the overridden definition in the derived class.

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** Modify VariantReturn.cpp to show that its behavior works with references as well as pointers.

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** In Early.cpp, how can you tell whether the compiler makes the call using early or late binding? Determine the case for your own compiler.

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. 25** Create a base class containing a clone( ) function that returns a pointer to a copy of the current object. Derive two subclasses that override clone( ) to return copies of their specific types. In main( ), create and upcast objects of your two derived types, then call clone( ) for each and verify that the cloned copies are the correct subtypes. Experiment with your clone( ) function so that you return the base type, then try returning the exact derived type.

**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. 27** Add a type called Tensor to OperatorPolymorphism.cpp.

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.** **30** (Advanced) If function calls to an object passed by value weren’t early-bound, a virtual call might access parts that didn’t exist. Is this possible? Write some code to force a virtual call, and see if this causes a crash. To explain the behavior, examine what happens when you pass an object by value.

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. 32** Size and no. of VPTRs.

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

**Ex. 34** Pretend that virtual functions don’t exist, and modify Instrument4.cpp so that it uses dynamic\_cast to make the equivalent of the virtual calls. Explain why this is a bad idea.

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);

**7: Exception Handling 7: Exception Handling**

**Ex. 1**

class Alfa { public:

class Bravo {public:

char c[5]={'W', '#', '\*', '$', '@'};

char\* kuh = c; };

void fina() throw (Bravo) { cout <<" Innen fina\n";

throw Bravo(); }

};

int main() {Alfa son; Alfa::Bravo lin;

try { son.fina(); }

catch(Alfa::Bravo) {

cout << lin.kuh << endl;

abort(); }

**Ex. 3**

int main() {

try {

//throw -9;

}

catch(...) {

cout <<"End?" << endl;

terminate(); }

**Ex.4**  Non functional**!**

class Alfa {static int i;

static const int sz=40; public:

Alfa() throw(int){ cout << "Konstru i:" << i << endl;

if(i==11) throw int(11);

++i; }

~Alfa() { cout << "Zerstoe :" << i-- << endl; }

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

cout << "Das Neue \n";

// for (int u = 0; u <=11; u++)

return ::new char[sz]; }

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

cout << "delete[] \n";

::delete []p; }

/\* static void fula(Alfa\* a1, int s) {

delete a1[s]; }\*/ };

int Alfa::i =1;

int main() { try { Alfa\* a1 = new Alfa[11]; }

catch (int 11) {

cout << " Run out of memory.\n"; }

**Ex.5**

class Alfa { public:

~Alfa() { throw 7;

cout << "Zerstoer" << endl; } };

int main() {

try {Alfa omega;

throw 8; }

catch(int) {

cout << "int kot \n"; } // Prints: terminate called after throwing

// an instance of ’int’. Aborted.

catch(Alfa) { cout << "Exter";}

**Ex. 6**

class Alfa { public:

~Alfa() { cout << "Wurde zerstoert! \n "; }

};

int main() { Alfa son;

try { throw son;

//son.fina(); }

catch(Alfa&) {

cout <<"End?" << endl; } //Prints: Wurde zerstoert! Two times

**Ex. 7**

class Alfa { public:

~Alfa() { cout << "Wurde zerstoert! \n "; }

};

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

try { throw Gulielmo;

}

catch(Alfa\*) {

cout << Gulielmo << endl;

// delete Gulielmo; } //The destructor is called if delete is explicitly used

cout <<Gulielmo << endl; /\*otherwise the destructor is not called.

What is the exact condition

For object to be cleaned-up? \*/

**Ex. 8**

class Alfa { string sn; public:

Alfa (string kui):sn(kui) { cout << "Alfa Konstruktor \n";

//cout << sn << endl;

}

Alfa(const Alfa&){ cout << "Alfa Kop-Konst. \n";

cout << sn ; } //Does not print sn.

~Alfa() {} };

int main() {

try {Alfa omega("Megaro");

//Alfa Renatte = omega;

throw omega; } // Copy constructor called

catch(Alfa&) {

cout << "Innen kot \n";

// terminate(); //Prints: terminate called after throwing an instance of 'Alfa'

}

**1: Strings 1: Strings 1: Strings**

**Ex. 2**

string strip0x0d(string suin) {

int i;

while ((i = suin.find("0x0d"))!= string::npos)

suin.erase(i,strlen("0x0d"));

return suin;

}

**Ex. 3 1st Method**

string todo(string ongo) {

const int u= ongo.size();

string suf= ongo;

for(int i= u; i>-1; i--)

suf[u-i-1]= ongo[i];

return suf;

}

int main() {

string cu("PINA COLLADA");

cout << cu.size() << " " << todo(cu) << endl;

**2nd Method**

string todo(string ongo) {

string sui;

string::reverse\_iterator evo;

for(evo = ongo.rbegin(); evo != ongo.rend(); evo++){

sui += \*evo;

return sui;

}

**Introduction to Templates Introduction to Templates**

**Ex.3** Modify TPStash.h so that the increment value used by inflate( ) can be changed throughout the lifetime of a particular container object.

template <class T, int incr> void PStash<T, incr>::inflate(int increase, int pzz = sizeof(T\*))

**Ex. 7** Modify AutoCounter.h so that it can be used as a member object inside any class whose creation and destruction you want to trace. Add a string member to hold the name of the class. Test this tool inside a class of your own.

#ifndef AUTOCOUNTER\_H

#define AUTOCOUNTER\_H

#include <set> // Standard C++ Library container

class AutoCounter {

static int count;

int id;

class CleanupCheck {

std::set<AutoCounter\*> trace;

public:

void add(AutoCounter\* ap) {

trace.insert(ap);

}

void remove(AutoCounter\* ap) {

require(trace.erase(ap) == 1,

"Attempt to delete AutoCounter twice");

}

~CleanupCheck() {

std::cout << "~CleanupCheck()"<< std::endl;

require(trace.size() == 0,

"All AutoCounter objects not cleaned up");

}

};

static CleanupCheck verifier;

AutoCounter() : id(count++) {

verifier.add(this); // Register itself

std::cout << "created[" << id << "]"

<< std::endl;

}

// Prevent assignment and copy-construction:

AutoCounter(const AutoCounter&);

void operator=(const AutoCounter&);

public:

// You can only create objects with this:

static AutoCounter\* create() {

return new AutoCounter();

}

void print() {

cout << "id = " << id << endl;

}

~AutoCounter() {

std::cout << "destroying[" << id

<< "]" << std::endl;

verifier.remove(this);

}

// Print both objects and pointers:

friend std::ostream& operator<<(

std::ostream& os, const AutoCounter& ac){

return os << "AutoCounter " << ac.id;

}

friend std::ostream& operator<<(

std::ostream& os, const AutoCounter\* ac){

return os << "AutoCounter " << ac->id;

}

};

int AutoCounter:: count = 0;

AutoCounter::CleanupCheck AutoCounter::verifier;

#endif

class Alfa { public:

AutoCounter\* ac1= AutoCounter::create();

};

int main() {

Alfa tori,lori;

tori.ac1->print();

lori.ac1->print();

delete tori.ac1;

delete lori.ac1;

**Ex.16** Using TStack2.h, TPStash2.h, and Shape.h, instantiate Stack and PStash containers for Shape\*, fill them each with an assortment of upcast Shape pointers, then use iterators to move through each container and call draw( ) for each object.

Testing code

template<class RO>

class Stack {

vector<RO> r;

};

class Shape { public:

/\* friend std::ostream& operator<<(std::ostream& οko, Shape& k) {

return oko << "Shape": << this;

}\*/

};

int main() {

vector<Shape> fifo;

Shape\* betizeiger;

for(int u = 0; u<5; u++){

betizeiger = new Shape;

fifo.insert(fifo.begin(), \*betizeiger);

}

cout << fifo.empty() << endl;

while(!fifo.empty()) {

\*betizeiger = fifo.front();

cout << betizeiger << endl;

fifo.erase(fifo.begin());

}

cout << (bool)(fifo.empty()) << endl;

return 0; }