Name	Matr.-Nr.

Exam Programming III – CTS

WS 17

Prof. Dr. K. Baer

Auxiliary materials: 1 page DIN A4, double sided

Preparation time: 90 Min.

1. Please enter name and matriculation number first!
2. Check the completeness of the task sheets.
3. The exam consists of 5 tasks. Get a quick overview of the tasks and get started on the task that will most likely bring you a sense of achievement.
4. Read the task carefully before you try to solve the task!
5. Use the space provided on the task sheets to answer the questions.
6. Write legibly. Unreadable parts are rated 0 points!

Good Luck!

Task	1	2	3	4	5	Sum
Points	23	8	16	10	33	90
achieved						

Task 1 (23 Points = 13+2+2+2+4)

The following class definition shall be given:

```
class Part {
public:
     Part()              { cout << " cPart"; }
     Part(const Part& a) { cout << " copyPart"; }
     ~Part()             { cout << " ~Part"; }
};

class Base {
private:
    Part p;
public:
    Base()                 { cout << " cBase";  }
    Base(const Base& b)    { cout << " copyBase";  }
    ~Base()                { cout << " ~Base"; }

    void method1(Base b)   { cout << " m1Base"; }
};

class Child : public Base {
private:
    Part* ptrP;
public:
    Child()                { cout << " cChild"; ptrP=0;}
    Child(const Child& c)  { cout << " copyChild"; ptrP=c.ptrP; }
    ~Child()               { cout << " ~Child"; if (ptrP) delete ptrP;}

    void method1(Base b)   { cout << " m1Child"; }
    void method1(Base* b)  { cout << " m1_Child"; b->method1(*b); }
    void method2()         { cout << " m2Child"; ptrP = new Part(); }
};
```

a) For the test () function below, after each line, specify which outputs appear on the console during evaluation.
Please write the word "NOTHING" in cases where you want to express that no no output is generated. Leaving a field blank means that you have not provided an answer and therefore will not get points for it.

No	Code	Output (write NOTHING if no outputis generated)
1	`void test(){`	
2	`Child c1;`	cPart cBase cChild
3	`Child c2 = c1;`	cPart cBase copyChild
4	`Base b1;`	cPart cBase
5	`Base* ptrB = &c2;`	NICHTS
6	`ptrB->method1(c1);`	cPart copyBase m1Base ~Base ~Part
7	`ptrB = new Child();`	cPart cBase cChild
8	`static_cast<Child*>(ptrB)→method2();`	m2Child cPart
9	`c1.method2();`	m2Child cPart
10	`delete ptrB;`	~Base ~Part
11	`Child* ptrC = &c1;`	NICHTS
12	`ptrC->method1(&c2);`	m1_Child cPart copyBase m1Base ~Base ~Part
13	`delete ptrC;`	~Child ~Base ~Part
14	`}`	~Base ~Part ~Child ~Base ~Part ~Child ~Base ~Part

b) Name the difference between overloading and overriding methods.
Which lines in the class definition of page 2 contain Overloading and which ones Overriding?

Methods with the same name:
Overload: method of the same name in the same class with different parameter list (lines 27 & 28)
Override: Method with the same name in different classes of an inheritance hierarchy with the same parameter list. (lines 16 & 27)

c) In row 10 and 21 of the class definitions (see page 2) different types of relationship between classes are used.
How do you call these two different types of relationships?
What are the consequences using these two types of modeling?

10: Composition → Lifetime dependency

21: Aggregation → Memory management must be solved, otherwise there is a risk of memory leakage

d) Where does a memory leak occur in method test ()? Explain why!

In line 8 Part-Object is created, which is not deleted again in line 10 when deleting prtB, because Destructor is not virtual.

In line 9 a new Part object is created, the old one creates a memory leak.

e) Now, the destructor of class *Base* as well as the method *method1* shall be declared as virtual.
Specify the changes in the output of *test()*.

6: cPart copyBase **m1Child** ~Base ~Part

8: m1_Child cPart copyBase **m1Child** ~Base ~Part

10: **~Child ~Part** ~Base ~Part

12: m1_Child cPart copyBase **m1Child** ~Base ~Part

Task 2 (8 points = 2 + 6)

a) In which cases should a programmer prefer `std :: list <T>` to `std :: vector <T>` and vice versa?

- frequent insert/delete in front or in the middle
- Index access not required

b) What problems may arise when applying explicit casts to derived classes ?

downcast: explicit type cast in class hierarchy downwards.
possible problem: Downcast is made, but underlying object is of base class type.
Example:

```
GeoObj* ptrGeo = new GeoObj(Koord(1,2));
Circle* ptrCircle = NULL;

ptrCircle = static_cast<circle*>(ptrGeo); //valid!
ptrCircle->draw();            // draw() of circle
ptrCircle→calcCircum();       // -->undefined

dynamic_cast:
ptrCircle = dynamic_cast<Circle*>(ptrGeo); // Null-Ptr!
if (ptrCircle == NULL) ... Error handling
        else ... normal processing
```

Task 3 (16 Points=4+6+6)

A game requires a class Player (see below), which essentially stores the score points and the time required. The players are managed in a scorelist of type vector <Player> (see test () - method).

Elementary functions in a game are:

- find the best player or
- find Players who are better or worse than a certain player.

```
class Player{
private:
      std::string name;
      int points;
      int time;
public:
      Player(std::string name, int p, int t) : name(name), points(p),
                                                time(t){}
      int getPoints() const { return points; }
      int getTime() const {   return time; }

      std::string toString() const {
            std::stringstream buffer;
            buffer << "Name: " << name << ", Points: " << points << ",
                       Time: " << time;
            return buffer.str();
      }
      bool operator>(const Player& other) const {
            return this->points > other.points;
      }
      bool operator<(const Player& other) const {
            return  this->points < other.points;
      }
};


std::ostream& operator<<(std::ostream& os, const Player& player){
      return os << player.toString() ;
}

void test(){
  std::vector<Player> scorelist;
  std::vector<Player> top;

  Player* Red = new Player("Red", 10, 15);
  Player* Purple = new Player("Purple", 20, 25);
  Player* Blue = new Player("Blue", 30, 25);
  Player* Yellow = new Player("Yellow", 40, 40);
```

```
  scorelist.push_back(*Red);
  scorelist.push_back(*Purple);
  scorelist.push_back(*Blue);
  scorelist.push_back(*Yellow);

  typedef std::vector<Player>::iterator Iter;
  std::ostream_iterator<Player>Output(std::cout, "\n");
  Iter start = scorelist.begin();
  Iter end = scorelist.end();

  Iter max = std::max_element( start, end,  CompareScore() ) ;  // TODO
  std::cout << (*max) << std::endl;

  my_copy_if(start, end, std::back_inserter(top),                // TODO
                  Compare<std::greater<Player>, Player&>         // TODO
                                  (std::greater<Player>(), *Purple) );
  copy(start, end, Output);
  copy(top.begin(), top.end(), Output);
}
```

You can use the STL function `max_element` to determine the largest element, if you have a corresponding comparison function.

a) Implement a functional object `CompareScore`, which compares players according to the formula: `2 * points / time`.

```
struct CompareScore{
        bool operator()(const Player& p1, const Player& p2){
                return (2*p1.getPoints()/(p1.getTime())) < (2*p2.getPoints()/
                                                                (p2.getTime()));
        }
};
```

b) With `my_copy_if`, all elements (players in the test-method above) that satisfy a condition (in the example above, those who are better than the player Purple) should be copied to a result container. Let us assume, that the function `copy_if` was forgotten in the STL, so this function has to be implemented.
As a parameter it requires:
   ◦ two iterators on the area in the source container from which values are

copied, if they fulfill the condition.
  - An iterator to the position in the result container, to which the results will be stored.
  - A binary function for comparing the values.
  - Return value is an iterator on the element after the last inserted element in the result container.

```
template<class InIter, class OutIter, class UnOp>
OutIter my_copy_if(InIter first, InIter last, OutIter dest, UnOp op){
	for(InIter i = first; i != last; ++i){
		if (op(*i)){
			*dest++  = *i;
		}
	}
	return dest;
}
```

c) Implement a template for a functional object `Compare`, which can be used in the example shown in method test (see above).

```
template<class Operation, class T>
class Compare //: public std::unary_function<T, bool> {
```

```
{
public:
    Compare(const Operation& o, const T& t) : op(o), y(t) {}
    bool operator()(const T& x) const {   return op (x,y);     }
private:
    const Operation op;
    const T y;
};
```

Task 4 (10 points=4+3+3)

a.) The following program code shall be given:

```
#include <iostream>

void g1(std::string& s)    { std::cout << s << std::endl; }
void g2(std::string* sptr){ std::cout << *sptr << std::endl; }

void f1(const std::string& s){
   g1(s);
   std::string localCopy = s;
   g2( &localCopy );
}

int main() {
   f1("Hallo");
   return 0;
}
```

Is the program compilable?

Explain the backgrounds and explain what happens in each of the three lines of method f1!

No.

g1 expects a reference to a string as parameter.

Theoretically, the string s could be changed in g1.

But f1 must not change s according to parameter declaration, therefore compile error.

Different with g2:

Here a non-constant copy of s is created before the call, which can be changed within f1. This has no external effect. Therefore the call g2 is allowed.

b.) What is the output of the following code fragment?
The values of *i and *r do not match. Why?

```
const int f[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
vector<int> v(f, f+10);
vector<int>::iterator i = v.begin() + 3;
vector<int>::reverse_iterator r(i);
cout << *i << " != " << *r << endl;
```

3 2

... because *r dereferences the previous element

c.) The following program will not be compiled. Why?

```
const int f[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
vector<int> v;
copy(f, f +10, front_inserter(v));
```

Vector --> push_front not available

front inserter cannot construct a corresponding inserter

Task 5 (33 points = 6+8+4+5+6+4)

Implement (**without** the use of the class vector of the standard template library) a new class `Point`, which should simplify the handling of coordinates. When creating an object of this class, it shall be possible to specify, how many dimensions are available.

Example:

```
Point p(1); // 1-dimensional, only a x-component
Point q(2); // 2-dimensional, q has a x and a y-component
Point r(3); // 3-dimensional, r has x,y and z-components
```

In order to be able to keep the dimension arbitrary, the components should be stored in an array which is requested at the time of object creation. The data type of the components is double.

a) Implement a class `Point` that offers this

b) In addition, access to a component shall be possible like the access to a single element of an array.

   Examples:

   ```
   p[0]  // x-component of point p
   q[1]  // y-component of point q
   r[2]  // z-component of point q
   ```

   If an attempt is made to access a component outside the dimension, the exception "`OutOfDimension`" shall be generated.

   The above component access mechanism should also allow the component values to be set or read so that, for example, the following accesses are possible:

   ```
   p[0] = 100.0;     // set x-component of p to 100.0
   p[1] = 0.0;       // set y-component of p to 0.0
   x[0] += p[0];     // add the value of the x-component of point p
                     // the x-component of point x
   ```

c) The construction of a 'point object' from another 'point object' shall be possible.

   Example:

   ```
   Point q(3);
   Point p = q;
   ```

d) The assignment of a point object to a (possibly different) point object only shall be possible if the dimension of the expression on the right side of the assignment is less than or equal to the dimension of the object on the left side. If the dimension of the object on the right side is really smaller, only the existing components will be copied.

Example:

```
Point p1(2), p2(2), q(3);
// ...
p1 = p2; // o.k.
q = p1;  // o.k., only the components of q contained in p1 will
         // be overwritten.
p2 = q;  // Error, since the dimension of q is three, but of p2
         // it is only two!
```

In the event of an error, the exception "`DimensionMismatch`" should be generated.

e) Develop an output operator for `Point`-objects. The output produced should look like: (100 5 42). The data encapsulation should not be affected by this!

f) In addition, develop a simple test program, which only has the task of creating and catching the two exceptions `DimensionMismatch` and `OutOfDimension` when using class `Point`. It is sufficient if the test program merely outputs an error message on the screen as error handling.

Hint:

The class is to be fully implemented, i.e. with all required constructors, destructors, etc. and possibly additional classes have to be added.

```
class Error  : public std::exception {
   private:
      std::string message;
   public:
      Error(const std::string& s) : message(s){};
      virtual ~Error() throw() {}
      const char* what() const throw()  { return message.c_str();}
                          // 4P
};

class Point{
```

```
private:
  double* components;
  int len;                                                       // 2 P
public:
  Point(int length):len(length){
    components = new double[len];
  }
          // 2 P

  Point(const Point& other){
    len = other.len;
    components = new double[len];
    for(int i = 0; i< len; ++i){
      components[i] = other.components[i];
    }
  }
          // 4 P

  virtual ~Point(){
    if (components) {
      delete[] components;
    }
  }
          // 2 P

  Point& operator=(const Point& other){                // 5 P
          if( this == &other){
                  return *this;
          }
    if(other.len > this->len){
      throw Error("Dimension mismatch ");
    }
    else{
      for(int i = 0; i< other.len; ++i){
        components[i] = other.components[i];
```

```
      }
    }
    return *this;
  }

  double& operator[](int dim){                                    // 4 P
    if (dim < 0 || dim > len){
        throw Error("Out Of Dimension");
    }
    return components[dim];
  }

  std::string toString() const {                             // 4 P
        std::stringstream buffer;
        buffer << "( ";
        for(int i = 0; i < len; ++i){
            buffer << components[i] << " ";
        }
        buffer << ")";
        return buffer.str();
  }
};

std::ostream& operator<<(std::ostream& os, const Point& p){
        os << p.toString();
        return os;
}                                                                 // 2 P
```

```
void test(){
        Point  x(1);    // 1-dimensional, entspricht: x hat nur eine x-Komponente
        Point  q(3);    // 3-dimensional, entspricht: q hat x,y und z-Komponenten

        try{
                Point  p = q;

                std::cout << x <<std::endl;
                std::cout << p <<std::endl;
                std::cout << q <<std::endl;

                p[0]  = 100.0;

                p[1]  =   5.0;  // y-Komponente von p auf 0.0 setzen
                x[0] +=  p[0];  // x-Komp. von p zur x-Komp. von x hinzuaddieren

                std::cout << x <<std::endl;
                std::cout << p <<std::endl;
                std::cout << q <<std::endl;

                Point  p1(2), p2(2), q(3);
                // ...
                p1 = p2;        // o.k.
                q  = p1;        // o.k. es werden nur die auch in p1 enthaltenen
                                   Komp. von q überschrieben.
                p2 = q;         // Fehler, da die Dimension von q drei, von p2
                                  jedoch nur zwei beträgt!
        }
        catch(std::exception& e){
                std::cout << e.what();
        }

                                // 4 P
}
```