LOICH KAMDOUM DEAMENI =>Mat-Nr: 506520
NDAME EKOBE HUGUE BENJAMIN => Mat-Nr: 495921

Uebungsblatt 5:

Ubung 1

1)
Size of Empty -> 1
Size of EmptyDerived -> 1
Size of NonEmpty -> 1

2)
- Wenn Empty keinen speicherplatz beansprüchen würde, dann hätten alle Objekte von der Klasse die gleiche Adresse im Speicher
- Im diesem Fall hätten c.a und c.b die gleiche Sprecheradressen.

3)
Size of Object Composite -> 8
Size of c.a -> 1
Size of c.b -> 4

4)
Size of Object CompositeChar -> 2
Size of c.a -> 1
Size of c.b -> 1

Size of Object CompositeLong -> 16
Size of c.a -> 1
Size of c.b -> 8

sizeof() returns the size, in bytes of an object. Die obigen Ergebnisse lassen sich erklären in sofern, dass die Grösse des Objekts die Grösse der Summe von alle seine Attribute gleich. Die Grösse eine Variable char (oder long long) ist 1 (oder 8) Byte in c++. In dem letzten Fall (long long) hätte die Grösse 9 sein gewesen, aber zur Optimierung der Speicher wird 8+8 in Spreicher reserviert.

Ubung_1.cpp

```
#include <iostream>

class Empty
{
};

class EmptyDerived : Empty
{
};

class NonEmpty : Empty
{
public:
    char c;
};
```

```
struct Composite
{
   Empty a;
   int b;
};

struct CompositeChar
{
   Empty a;
   char b;
};

struct CompositeLong
{
   Empty a;
   long long b;
};

int main()
{
   Empty e1;
   EmptyDerived e2;
   NonEmpty e3;

   std::cout << "Size of Empty : " << sizeof(e1) << std::endl;
   std::cout << "Size of EmptyDerived : " << sizeof(e2) << std::endl;
   std::cout << "Size of NonEmpty : " << sizeof(e3) << std::endl;

   Composite c1;
   std::cout <<"Adresse von c.a : " << &c1.a << std::endl;
   std::cout <<"Adresse von c.b : " << &c1.b << std::endl;

   std::cout << "Size of Object Composite : " << sizeof(c1) << std::endl;
   std::cout << "Size of c.a : " << sizeof(c1.a) << std::endl;
   std::cout << "Size of c.b : " << sizeof(c1.b) << std::endl;

   CompositeChar c2;
   std::cout << "Size of Object CompositeChar : " << sizeof(c2) << std::endl;
   std::cout << "Size of c.a : " << sizeof(c2.a) << std::endl;
   std::cout << "Size of c.b : " << sizeof(c2.b) << std::endl;

   CompositeLong c3;
   std::cout << "Size of Object CompositeLong : " << sizeof(c3) << std::endl;
   std::cout << "Size of c.a : " << sizeof(c3.a) << std::endl;
   std::cout << "Size of c.b : " << sizeof(c3.b) << std::endl;

   return 0;
}
```

Übung 2 Vererbung und Komposition

* Erste Variante

NumVector.cpp

```
#include <iostream>
#include "NumVector.h"

NumVector::NumVector(int size)
{
   this->size = size;
   this->cachedNorm = -1;
   this->values = new double[size];
}

NumVector::~NumVector()
{
   this->size = 0;
   this->cachedNorm = 0;
   delete [] values;
}

double &NumVector::operator[](int index)
{
   return this->values[index];
}

const double &NumVector::operator[](int index) const
{
   return this->values[index];
}

double NumVector::norm()
{
   if(this->cachedNorm != -1)
      return this->cachedNorm;

   double result = 0;
   for(int i = 0; i < this->size; ++i)
      result += this->values[i] * this->values[i];

   this->cachedNorm = sqrt(result);
   return this->cachedNorm;
}

const double NumVector::norm() const
{
   if(this->cachedNorm != -1)
      return this->cachedNorm;

   double result = 0;
   for(int i = 0; i < this->size; ++i)
      result += this->values[i] * this->values[i];

   this->cachedNorm = sqrt(result);
   return this->cachedNorm;
```

```
}

void NumVector::resize(int size)
{
   double *tmp = new double[size];
   for(int i = 0; i < size; ++i)
     tmp[i] = values[i];
   delete [] values;
   values = new double[size];
   for(int i = 0; i < size; ++i)
     values[i] = tmp [i];
   delete [] tmp;
}
```

* Zweite Variante

NumVector.cpp

```
#include <iostream>
#include "NumVector.h"

NumVector::NumVector(int size)
{
   this->size = size;
   this->cachedNorm = -1;
   this->values = new double[size];
}

NumVector::~NumVector()
{
   this->size = 0;
   this->cachedNorm = 0;
   delete [] values;
}

double &NumVector::operator[](int index)
{
   return this->values[index];
}

const double &NumVector::operator[](int index) const
{
   return this->values[index];
}

double NumVector::norm()
{
   if(this->cachedNorm != -1)
      return this->cachedNorm;

   double result = 0;
   for(int i = 0; i < this->size; ++i)
      result += this->values[i] * this->values[i];
```

```
    this->cachedNorm = sqrt(result);
    return this->cachedNorm;
}

const double NumVector::norm() const
{
    if(this->cachedNorm != -1)
        return this->cachedNorm;

    double result = 0;
    for(int i = 0; i < this->size; ++i)
        result += this->values[i] * this->values[i];

    this->cachedNorm = sqrt(result);
    return this->cachedNorm;
}

void NumVector::resize(int size)
{
    double *tmp = new double[size];
    for(int i = 0; i < size; ++i)
      tmp[i] = values[i];
    delete [] values;
    values = new double[size];
    for(int i = 0; i < size; ++i)
      values[i] = tmp [i];
    delete [] tmp;
}
```

NumVector.h

```
#ifndef NUMVECTOR_H
#define NUMVECTOR_H
#include <math.h>

class NumVector
{
public:
    NumVector(int size);
    ~NumVector();
    double &operator [] (int);
    const double &operator [] (int) const;
    double norm();
    const double norm() const;
    void resize(int);

private:
    int size;
    mutable double cachedNorm;
    double *values;
};

#endif // NUMVECTOR_H
```

Ubung_2.cpp

```
#include<vector>
#include<iostream>
#include<cmath>
#include "NumVector.h"

int main(){
NumVector v(3);
v[0]=1; v[1]=3, v[2]=4;
const NumVector& w=v;
std::cout<<"norm is "<<w.norm()<<std::endl;
std::cout<<"vector is ["<<w[0]<<", "<<w[1]<<", "<<w[2]<<"]"<<std::endl;
}

// void vectorTest(std::vector<double>& v){}
// int main(){
// NumVector v(3);
// v.resize(2); // Darf wie andere std::vector Funktionen nicht sichtbar sein!
// vectorTest(v); // Hier muss auch ein Compiler Fehler auftreten!
// }
```

Übung 3 Vererbung und Funktionen

1)

Funktioniert nicht, weil C auf A nicht zugegriffen kann, da die Vererbung von B privat ist. Es wird versucht über die Funktion test() auf die Methode A::foo() zuzugreifen, da diese nicht in C definiert ist.

Zur Lösung muss die Vererbung in der Klasse C von B in "public" umgewandelt werden.

Dann kriegen wir im Ausgabe:

```
A::foo
B::foo
A::foo
B::foo
A::foo
```

Erklärung:

Zeile 1: foo() ist eine Funktion der Klasse A.
Zeile 2: B hat seine eigene Funktion foo(), deswegen wird seine aufgeruft und nicht die von A.
Zeile 3: B hat seine eigene Funktion foo(), aber der Parameter wird in ein Objekt der Klasse A umgewandelt, deswegen wird die Funktion foo() von A aufgeruft.
Zeile 4: C hat eine Funktion foo(), aber die Funktion test() benutzt ein Objekt von A, So die Funktion A::foo() wird durch die Vererbung (public) von B erreicht.

Ubung_3.cpp

```
#include <iostream>

class A
{
public:
   void foo() const
```

```
    {
        std::cout<<"A::foo"<<std::endl;
    }
};

class B: public A
{
public:
    void foo()
    {
        std::cout<<"B::foo"<<std::endl;
    }
};

class C: public B
{
public:
    void bar()
    {
        foo();
    }
    friend void test(const A&);
};

void test(const A& a)
{
    a.foo();
}

int main()
{
    A a; B b; C c;
    a.foo();
    b.foo();
    test(b);
    c.bar();
    test(c);
    return 0;
}
```

Übung 4 Exceptions und Destruktoren

Foo.cpp

```
#include "my_exception.cpp"

class Foo
{
public:
  ~Foo()
  {
    throw my_exception("Foo exception");
  }
};
```

```
class Bar
{
public:
  Bar()
  {
    throw my_exception("Bar exception");
  }
};

int main(int argc, char const *argv[]) {
  try {
    Foo f;
    Bar b;
  }
  catch (const std::exception & e){
    std::cout << "ERROR:" << e.what() << std::endl;
  }
  return 0;
}
```

my_exception.cpp

```
#include "my_exception.h"

void my_exception::my_exception(std::string msg_)
{
  msg = msg_;
}

virtual const char* my_exception::what() const noexcept
{
  return this->msg.c_str();
}
```

my_exception.h

```
#include <iostream>
#include <string>
#include <exception>
#include <utility>

class my_exception
{
private:
  std::string msg = "";

public:
  void my_exception(std::string);
  virtual const char* what() const noexcept;
};
```

Übung 5 Exceptions und Sanity Checks

NumVector.cpp

```
#include <iostream>
#include "NumVector.h"

NumVector::NumVector(int size)
{
   this->size = size;
   this->cachedNorm = -1;
   this->values = new double[size];
}

NumVector::~NumVector()
{
   this->size = 0;
   this->cachedNorm = 0;
   delete [] values;
}

double &NumVector::operator[](int index)
{
   if(index < 0 || index >= this->size)
      std::cout << "Error: Index: " << index << " is out of Array " << std::endl;

   return this->values[index];
}

const double &NumVector::operator[](int index) const
{
   if(index < 0 || index >= this->size)
      std::cout << "Error: Index: " << index << " is out of Array " << std::endl;

   return this->values[index];
}

NumVector operator * (const NumVector &v1, const NumVector &v2)
{
   NumVector tmp(v1.size);
   if(v1.size != v2.size)
      std::cout << "Error: Incompatibel Length " << std::endl;
   else
     for(int i = 0; i < v1.size; ++i)
      tmp[i] = v1.values[i]*v2.values[i];

   return tmp;
}

double NumVector::norm()
{
   if(this->cachedNorm != -1)
      return this->cachedNorm;

   double result = 0;
   for(int i = 0; i < this->size; ++i)
```

```
        result += this->values[i] * this->values[i];

    this->cachedNorm = sqrt(result);
    return this->cachedNorm;
}

const double NumVector::norm() const
{
    if(this->cachedNorm != -1)
        return this->cachedNorm;

    double result = 0;
    for(int i = 0; i < this->size; ++i)
        result += this->values[i] * this->values[i];

    this->cachedNorm = sqrt(result);
    return this->cachedNorm;
}

void NumVector::resize(int size)
{
    double *tmp = new double[size];
    for(int i = 0; i < size; ++i)
      tmp[i] = values[i];
    delete [] values;
    values = new double[size];
    for(int i = 0; i < size; ++i)
      values[i] = tmp [i];
    delete [] tmp;
}
```

NumVector.h

```
#ifndef NUMVECTOR_H
#define NUMVECTOR_H
#include <math.h>

class NumVector
{
public:
    NumVector(int size);
    ~NumVector();
    double &operator [] (int);
    const double &operator [] (int) const;
    friend NumVector operator * (const NumVector &, const NumVector &);
    double norm();
    const double norm() const;
    void resize(int);

private:
    int size;
    mutable double cachedNorm;
    double *values;
```

```
};

#endif // NUMVECTOR_H
```

Ubung_5.cpp

```
#include<vector>
#include<iostream>
#include<cmath>
#include "NumVector.h"

int main(){
  NumVector v(3);
  NumVector v2(2);
  v[0]=1; v[1]=3, v[2]=4;
  v2[0]=1; v2[1]=3, v2[2]=4;
  const NumVector& w=v;
  const NumVector& w2=v*v;
  const NumVector& w3=v*v2;
  std::cout<<"norm is "<<w.norm()<<std::endl;
  std::cout<<"vector is ["<<w[0]<<", "<<w[1]<<", "<<w[2]<<", "<<"]"<<std::endl;
  std::cout<<"vector is ["<<w2[0]<<", "<<w2[1]<<", "<<w2[2]<<", "<<"]"<<std::endl;
}
```