3. Semester : 2. Prüfung

Name :	

- Die gesamte Prüfung bezieht sich auf die Programmierung in C++ !!
- Prüfungsdauer: 90 Minuten
- mit Kugelschreiber oder Tinte schreiben
- Lösungen können direkt auf die Aufgabenblätter geschrieben werden
- PCs sind nicht erlaubt
- Unterlagen und Bücher sind erlaubt
- Die Aufgaben sind in Fragen unterteilt, die mit Kleinbuchstaben gekennzeichnet sind
- Achte auch auf Details wie Punkte oder Kommas und Semikolons. !!!!!!
- In den Beispielen fehlt meistens folgender Code, der als gegeben gilt:

```
#include <iostream>
using namespace std;
int main()
{
....
    return 0;
}
```

Aufgabe	**Punkte**	
Grundlagen	13	
Objektorientierte Programmierung	10	
Dynamischer Speicher und Klassen	10	
Vererbung und Polymorphismus	10	
Weitere Fragen	8	
TOTAL	51	

1. Grundlagen

a) Finde die richtigen Variablendeklarationen, also solche die keine Compilerfehler erzeugen und bezeichne diese mit einem Haken. (4 Punkte)

Beachte vor allem die Datentypen!!

Deklaration	**Richtig ?**
float note = -4.5;	ok
char* zahl = “1000”; **Beachte dass es ein STRING ist**	ok
hallo = “Hallo”; **Datentyp fehlt !**	
char A = 65; **ist auch ok, 65 ist gleich ‘A’ ASCII Code**	ok
int k = 10 **Semikolon fehlt**	
int z[] = { 100, 100, 100, 100}; **absolut OK**	ok

b) Definiere einen Aufzählungstypen als enum mit dem Namen Farben und den Elementen Rot, Grün und Blau. (4 Punkte)

```
enum Farben
{
    Rot,
    Gruen,
    Blau
};
```

Sollte eigentlich klar sein ! Der Wert von Rot ist übrigens 0, der von Gruen 1 und der von Blau 2. Diese Werte können bei einem enum aber explizit gesetzt werden ! Lies sonst nach im Skript! Umlaute (ä, ö, ü) sind in C++ nicht erlaubt !

c) Definiere eine Konstante “PI” mit dem Wert 3.1415. (2 Punkte)

const double PI = 3.1415;

const als wichtiges Schlüsselwort. float als Datentyp wäre auch ok. Genau an die Anweisungen halten, wenn da steht 3.1415 meine ich nicht 3.14152... ! SEMIKOLON !

d) Gegeben sei folgender Code mit den zwei selbstgeschriebenen Swap-Funktionen Swap1 und Swap2 :

```
void Swap1(int* pWert1, int* pWert2)
{
  int z = *pWert1;
  *pWert1 = *pWert2;
  *pWert2 = z;
}

void Swap2(int& wert1, int& wert2)
{
  int z = wert1;
  wert1 = wert2;
  wert2 = z;
}

int main()
{
  int a = 10;
  int b = 20;

  // hier Funktionsaufruf für Swap1
  Swap1(&a, &b);

  // hier Funktionsaufruf für Swap2
  Swap2(a, b);


  return 0;
}
```

Ergänze im Code oben die zwei fehlenden Zeilen für den Aufruf der Funktionen Swap1 und Swap2 (2 Punkte). Was haben die Variablen a und b am Ende des Programms für einen Wert ?(1 Punkt) (Total 3 Punkte)

a = 10, b = 20

Durch zweimaligen Aufruf einer Swap-Funktion sind die Variablen wieder wie zu Beginn des Programms. Wichtig ist hier die Parameterübergabe mittels Zeiger bei Swap1 oder mittels Referenz bei Swap2. Bei der Übergabe mit Referenz muss der aufrufende Code nicht speziell aussehen. Bei der Übergabe mittels Zeiger bei Swap1 muss die Adresse aber mit dem Adress-Operator & ermittelt werden.

2. Objektorientierte Programmierung

a) Hier eine Klasse DancyDinky (seufz).

```
#include <string>
#include <iostream>

using namespace std;

class DancyDinky
{
  public:
    DancyDinky(const char* name); // Konstruktor mit Par.
    ~DancyDinky();  //Destruktor
    void Tanze(int wievielmal);
  private:
    string m_name;
};
// Dem Konstruktor einfach den Namen als string mitgeben
DancyDinky::DancyDinky(const char* name)
{
  m_name = name;
  cout << "Hallo ich bin " << m_name << endl;
}

DancyDinky::~DancyDinky()
{
  cout << m_name << " geht jetzt" << endl;
}

void DancyDinky::Tanze(int wievielmal)
{
  for(int i = 0; i < wievielmal; ++i)
  {
    cout << m_name << " tanzt jetzt" << endl;
  }
}
```

Schreibe eine vollständige main-Funktion, die durch **Verwendung der Klasse DancyDinky** folgende Ausgabe erzeugt (Der Code ist nicht mehr als 5~6 Zeilen lang sein) : (4 Punkte)

```
Hallo ich bin Sepp
Sepp tanzt jetzt
Sepp tanzt jetzt
Sepp tanzt jetzt
Sepp tanzt jetzt
Sepp tanzt jetzt
Sepp geht jetzt
```

```
int main()
{
   DancyDinky seppDinky(“Sepp”);
   seppDinky.Tanze(5);
   return 0;
}
```

Hier gilt es die gegebene Klasse zu verstehen um sie danach im main wie gewünscht anzuwenden. Dem Konstruktor mit Parametern kann ein char* mitgegeben werden. Die einfachste Art dies zu tun seht ihr in der Lösung, denn ein “sjfsdjfkjsk” ist ein char* :
char* test = “sdkflasdfmkdkl”;
Dieser String wird dann im Konstruktor im Datenelement m_name von DancyDinky gespeichert. Zusätzlich wird der Begrüssungstext ausgegeben.
Die Funktion Tanze hat als Parameter einen int, der angibt, wie häufig das DancyDinky-Objekt tanzt.
Der Destruktor wird automatisch aufgerufen.
Hätte ich in der Aufgabe die Ausgabe so gestaltet, dass der Destruktor nicht aufgerufen werden sollte, hättet ihr das Objekt dynamisch mit new erzeugen müssen. Durch weglassen von delete hätte man dann verhindern können, dass der Destruktor aufgerufen wird. Hier hingegen wird der Destruktor automatisch aufgerufen, wenn das Objekt am Ende der Funktion weggeräumt wird.

b) Deklariere eine Klasse, die einen Schüler (Schueler) abstrahiert.
Ein Schüler hat einen Namen, ein Alter, und eine Durchschnittsnote. Diese Klasse soll einen Konstruktor haben, dem als Parameter der Name übergeben wird. Zusätzlich soll die Klasse zwei Funktionen haben, mit denen jeweils das Alter (SetzeAlter) und die Note gesetzt werden kann (SetzeNote). Wie gesagt : Es genügt die Deklaration der Klasse, der Code zu den Funktionen ist nicht gefragt. Deklariere die Datenelemente mit den passenden Datentypen im privaten Teil der Klasse ! (Vergiss nicht die Semikolons) (6 Punkte).

```
class Schueler
{
   public:
       Schueler(const string& name);
       void SetzeAlter(int alter);
        void SetzeNote(double note);
   private:
        string m_name;
         int m_alter;
         double m_note;
};
```

Haltet Euch hier genau an das in der Aufgabe geforderte ! Es lassen sich im Notfall auch Teilpunkte erreichen. Wichtig ist auch hier die richtige Wahl der Datentypen. Eine Note hat nun einmal Kommastellen (auch wenn ihr alle einen 6er macht !), also double (oder float). Haltet Euch auch daran die Datenelemente im privaten Teil der Klasse zu deklarieren auch wenn ich es nicht explizit hinschreibe (sonst werde ich wild !) Es ist Euch auch erlaubt die string Klasse zu verwenden. Ihr könnt zur Sicherheit noch den
#include <string>
using namespace std;
Teil hinschreiben !
Damit ihr länger mit der Prüfung habt, werde ich verlangen, dass ihr die Definition, also den Code für die Funktionen hinschreibt !! (Dafür gibts ein paar Punkte mehr !)

3. Dynamischer Speicher und Klassen

a) Ergänze in folgender Klassedeklaration und -definition den Destruktor (2 Punkte) und den Code im Konstruktor, der alle Elemente im Array auf 0 initialisiert (2 Punkte). (Total 4 Punkte)

```
class Leck
{
public:
  Leck();
  // HIER DESTRUKTOR DEKLARIEREN
  ~Leck();


private:
  int* m_pData;
};

Leck::Leck()
{
  m_pData = new int[20];
  // HIER DATEN IM ARRAY AUF 0 INITIALISIEREN
  for(int i = 0; i < 20; ++i)
  {
    m_pData[i] = 0;
  }




}

// HIER KORREKTE DESTRUKTORDEFINITION EINFÜGEN

Leck::~Leck()
{
  delete [] m_pData;
}
```

Hier finden wir folgende Elemente, die ihr einfach können müsst (und auch könnt) :
- Einen Destruktor deklarieren und definieren
- **eine korrekte Schleife mit einer festen Anzahl (hier 20) Wiederholungen**
- **Zugriff auf Elemente im Array mit den [] (eckigen Klammern, auch Index-operator genannt).**
- dynamische Felder erzeugen und löschen (z.B. int* pN = new int [xxxx]; delete [] pN;)

b) Ergänze in der Klasse unten den Code für den Destruktor (2 Punkte) und den Kopierkonstruktor (4 Punkte). (Total 6 Punkte)

```
#include <string.h>
class Student
{
public:
  Student(const char* name);// Konstruktor
  Student(const Student& c);// Kopierkonst.
  ~Student(); // Destruktor
private:
  char* m_pName;
};

Student::Student(const char* name)
{
  int laenge = strlen(name);
  m_pName = new char[laenge+1];
  strcpy(m_pName, name);
}

Student::~Student()
{
  delete []m_pName;
}

Student::Student(const Student& c)
{
  int laenge = strlen(c.m_pName);
  m_pName = new char[laenge+1];
  strcpy(m_pName, c.m_pName);
}
```

Hier geht es darum zuerst das zu machen was einfach ist um sich einige Punkte sofort zu sichern. Hier ist es den Destruktor hinschreiben ! Und zwar auch wenn ich es nicht explizit in dieser Reihenfolge in der Aufgabenstellung hinschreibe ! Nachher könnt Ihr in Ruhe überlegen, wie dieser Kopierkonstruktor zu lösen ist. Dabei hilft der Konstruktor sehr, denn der Code ist fast der gleiche. Im Konstruktor wird aber der char* direkt mitgegeben was beim Kopierkonstruktor nicht so ist. Dort steckt der char* aber als Datenelement des zu kopierenden Objektes im Objekt, das im Beispiel "c" heisst. Was wichtig ist, es genügt nicht nur den m_pName gleich den m_pName von c zu setzen !
m_pName = c.m_pName;
Denn da wäre das Objekt keine Kopie sondern würde den Speicherbereich auf den m_pName zeigt teilen. Dies könnte FATAL sein, denn wenn ein Objekt den Inhalt ändert oder den Zeiger löscht (siehe Destruktor) zeigt der Zeiger des anderen Objektes auf irgendwas, was womöglich gelöscht ist ! Das Programm würde abstürzen. Es muss also der Inhalt des Speichers kopiert werden !

4. Vererbung und Polymorphismus

a) Was gibt dieses Programm aus ? (6 Punkte)

```
class Basis
{
public:
  Basis();
  ~Basis();
};

class Abgeleitet : public Basis
{
public:
  Abgeleitet();
  ~Abgeleitet();
};

Basis::Basis()
{
  cout << "Basis kommt" << endl;
}

Basis::~Basis()
{
  cout << "Basis geht" << endl;
}

Abgeleitet::Abgeleitet()
{
  cout << "Abgeleitet kommt" << endl;
}

Abgeleitet::~Abgeleitet()
{
  cout << "Abgeleitet geht" << endl;
}

int main()
{
  Basis* p = new Abgeleitet;
  delete p;
  return 0;
}
```

Basis kommt
Abgeleitet kommt
Basis geht

Hier heisst es zu verstehen in welcher Reihenfolge die Konstruktoren bei Ableitung aufgerufen werden. Zuerst Basis danach Abgleitet. Der Zeiger der mit “delete” gelöscht wird ist vom Typ Basis. Da der Destruktor nicht virtuell (virtual) ist wird nur der Destruktor von Basis aufgerufen. Wäre dieser virtuell würden beide aufgerufen werden : zuerst Abgeleitet danach Basis. Wäre der Zeiger vom Typ Abgeleitet würden auch beide Destruktoren aufgerufen werden. Hier aber nicht, was häufig nicht gut ist (bei dynamischem Speicher z.B.)

b) Betrachte den folgenden Code (Code geht bis auf nächste Seite !) :

```
#include <iostream>
using namespace std;

class Basis
{
public:
  Basis();
  virtual void Arbeite();
};

class Abgeleitet : public Basis
{
public:
  Abgeleitet();
  // HIER ERGÄNZEN
  virtual void Arbeite();


};

Basis::Basis()
{
  cout << "Basis kommt" << endl;
}

void Basis::Arbeite()
{
  cout << "Basis arbeitet" << endl;
}

Abgeleitet::Abgeleitet()
{
  cout << "Abgeleitet kommt" << endl;
}

// HIER ERGÄNZEN

void Abgeleitet::Arbeite()
{
  cout << “Abgeleitet arbeitet” << endl;
}
```

```
int main()
{
  Basis* p = new Abgeleitet;
  p->Arbeite();
  delete p;
  return 0;
}
```

Ergänze den Code an den beiden angegebenen Stellen so, dass das Programm folgenden Text ausgibt :

```
Basis kommt
Abgeleitet kommt
Abgeleitet arbeitet
```

Die main-Funktion, sowie die Klasse Basis darf nicht geändert werden ! (4 Punkte)

Hier besitzt das main-Programm nur einen Zeiger auf die Basisklasse. Da aber die Funktion "Arbeiten" virtuell ist, können wir diese einfach in der abgeleiteten Klasse überschreiben. Beim Aufruf wird dann diese überschriebene Version aufgerufen, denn das Objekt, das sich hinter dem Zeiger p versteckt ist ein Objekt der Klasse "Abgeleitet".
Beachte, dass beim Konstruktor der Konstruktor der Basisklasse auch aufgerufen wird und beim virtuellen Destruktor auch beide aufgerufen werden. Bei anderen, normalen Funktionen wird jedoch nur die Version der abgeleiteten Klasse aufgerufen und nicht zusätzlich die Funktion der Basisklasse... !

5. Weitere Fragen

a) Welches Tool, das wir kennengelernt haben, bietet die Möglichkeit durch simples drag and drop eine grafische Benutzeroberfläche zu erstellen. (1 Punkt)

Borland **Die Borland-IDE muss irgendwo mindestens erwähnt werden ;-)**

b) Wie heissen die beiden Schlüsselworte, die verwendet werden um dynamischen Speicher zu verwalten ? (2 Punkte)

new, delete **Mit new wird dynamisch Speicher alloziert, mit delete wieder freigegeben.**

c) Zeichne ein kleines Klassendiagramm mit den Beziehungen zwischen den Klassen, wie wir das bereits früher gemacht haben und zwar für folgende Klassen : Person, Student, Lehrer und Schulangehoeriger. (4 Punkte)

d) Willst Du fehlerfreie C++ - Programme schreiben ? (1 Punkt) **JAA.**
Das ist eine Glaubensfrage ;-) Jedesmal wenn ihr ab jetzt Programmiert denkt ihr an dieses "Versprechen" und macht alles um Fehler zu vermeiden.