Hochschule Coburg
Studiengang Informatik

Name: Sabuncuoglu, Enes

BACHELORPRÜFUNG

Prüfungsfach: **Fortgeschrittene Programmierung**

Prüfungstag: 26. Januar 2021

Reine Arbeitszeit: 90 Minuten
Erlaubte Hilfsmittel: keine

Aufgabe 1: Python: Datentypen von Variablen (10 Punkte)

Was ist der Wert und Typ folgender Python-Ausdrücke?

Ausdruck	Wert/Ausgabe	Typ
`9//2/2`	2.0	float
`(~3+1)<<2`	-12	int
`[k%2 for k in range(5)]`	[0,1,0,1,0]	list
`[a+a for a in "abc"]`	["aa","bb","cc"]	list

Aufgabe 2: Python: Listen und Tupel (8 Punkte)

a) Gegeben sei eine Liste q mit Zahlen. Erstellen Sie eine neue Liste, die nur diejenigen Elemente aus q enthält, die sowohl gerade Zahlen sind als auch an einer geraden Position (Index) in der Liste stehen.

Beispiel: Für die Liste `q = [ 1, 3, 5, 8, 10, 13, 18, 36, 78]` soll die Ausgabe `[10, 18, 78]` lauten.

```
q1=[q[i] for i in range(len(q)) if i%2==0 and q[i]%2==0]
```

b) Schreiben Sie einen Lambda-Ausdruck, der eine Funktion mit Parameter n darstellt, deren Rückgabewert eine Liste aller Vielfachen von 3 zwischen 1 und n ist. Damit soll folgende Ausgabe erzeugt werden:

```
>>> z = list(map(lambda                    , [13,7]))
>>> print(z)
[[0, 3, 6, 9, 12], [0, 3, 6]]
```

```
lambda i:[x for x in range(i) if x%3==0]
```

Aufgabe 3: Python: Klassen und dynamische Datenstrukturen (12 Punkte)

Es soll eine Python-Klasse implementiert werden, die einen Stack repräsentiert. Diese soll über folgende Inhalte verfügen:

a) Ein privates Attribut vom Typ `list`, das die Inhalte des Stacks verwaltet

b) Ein Konstruktor zur Initialisierung mit einer leeren Liste

c) Eine Methode `push()`, um einen Eintrag in den Stack einzufügen

d) Eine Methode `pop()`, die den letzten Eintrag wieder ausgibt und ihn aus der Liste löscht (mit dem Befehl `del`)

e) Eine Methode `empty()`, die prüft, ob Elemente im Stack vorhanden sind

Damit soll folgender Code realisierbar sein:

```
>>> wort = input("Wort: ")
>>> stapel = Stack()
>>> for zeichen in wort:
:    stapel.push(zeichen)
>>> while not stapel.empty():
:    print(stapel.pop(), end=" ")
```

Wort: Klausur

r u s u a l K

```
class Stack:
    def __init__():
        self.__stack() = []

    def push(item):
        self.__stack.append(item)
    def pull():
        if not self.empty():
            self.__stack.pop()
        else
            print("Stack is empty")
            return None
    def empty(self):
            return len(self.__stack) == 0
```

Aufgabe 4: C++: Klassen, Konstruktoren und Destruktoren, Vererbung (41 Punkte)

Mit Hilfe des Entwurfsmusters „Strategie“ soll eine Möglichkeit geschaffen werden, Zeichenketten mit unterschiedlichen Verfahren zu serialisieren. Die Verfahren verfügen dabei über verschiedene Systemparameter (z.B. Codierungsverfahren), die ebenfalls in Klassen repräsentiert werden sollen.

```
01  class Parameter
02  {
03  private:
04    char*        encoding;
05    unsigned int enc_size;   // Länge des encoding-Arrays
06  public:
07    Parameter();
08    Parameter(const char* enc, unsigned int esize);
09    Parameter(const Parameter& p);
10    ~Parameter();
11    // …
12  };

13  class Serialisation
14  {
15  private:
16    Parameter* params;
17  public:
18    Serialisation(Parameter* p = nullptr);
19    ~Serialisation();
20    virtual int serialize(const char* source, char* dest, int size);
21    virtual int deserialize(const char* source, char* dest, int size);
22  };
```

a) Die Klasse `Parameter` benötigt einen Kopierkonstruktor, da sie über dynamisch verwalteten Speicher verfügt.

i. Warum benötigt die Klasse Parameter zusätzlich noch einen Zuweisungsoperator?

Der Zuweisungsoperator ist erforderlich, damit ein bereits existierendes Objekt der Klasse Parameter mit einem neuen Wert überschrieben werden kann, ohne dass es zu Speicherverlusten oder Inkonsistenz kommt.

ii. Geben Sie eine Implementierung des Zuweisungsoperators nach dem **copy&swap-Idiom** an. Gehen Sie davon aus, dass der Kopierkonstruktor bereits implementiert ist.

```
Parameter& operator=(Parameter other) {
    swap(*this, other);
    return *this;
}
friend void swap(Parameter& first, Parameter& second) {
    using std::swap;
    swap(first.encoding, second.encoding);
    swap(first.enc_size, second.enc_size);
```

b) Zur Klasse `Parameter` soll ein Movezuweisungsoperator hinzugefügt werden.

i. Was ist der Unterschied zwischen einem einfachen Zuweisungsoperator und einem Movezuweisungsoperator?

Ein Zuweisungsoperator kopiert die Datenfelder in ein anderes Objekt der Klasse. Dies erfordert aber eine Kopie aller Daten aus dem aktuellen Objekt, was Zeit & Speicherplatz frisst. Das ist insbesondere bei großen Objekten mit viel dynamischen Speicher schlecht. Dementgegen steht der Movezuweisungsop., welcher die Eigentümerschaft des dynamischen Speichers & anderer Ressourcen auf das Zielobjekt überträgt. Das Quellobjekt wird dann anschließend in einen leeren Zustand versetzt. Dies ist in der Regel schneller & speichereffizienter als eine Kopie, da keine tatsächliche Kopie der Daten erforderlich ist.

ii. Schreiben Sie eine Implementierung für diesen Operator.

```
Parameter& operator=(Parameter&& other){
    if(this != &other){
        delete[]encoding
        encoding = other.encoding;
        enc_size = other.enc_size;
        other.encoding = nullptr;
        other.enc_size = 0;
    }
    return *this;
}
```

nur Referenz auf temporäre Objekte

iii. Vervollständigen Sie folgenden Codeausschnitt so, dass dieser Operator aufgerufen wird.

```
Parameter p1 ("test", sizeof(test));
Parameter p2;
p2 = std::move(p1);
```

c) Die Klasse `Serialisation` soll die Schnittstelle für die verschiedenen Serialisierungsverfahren werden.

i. Wie ist die Klasse zu ändern, damit die Methoden `serialize` und `deserialize` rein virtuell sind? Welche Auswirkungen hat das?

```
virtual int serialize(const char* source, const char* dest, int size) =0;
virtual int deserialize(const char* source, const char* dest, int size) =0;
```

1. Nicht mehr möglich ein Objekt der Klasse Serialisation zu erstellen, da keine Implementation für die Methoden gibt.
2. Ermöglicht, dass abgeleitete Klassen ihre eigene Implementierung für diese Methoden bereitstellen was zu polymorphismus führt.
3. Optimal für abstrakte Schnittstellen mit verschiedenen implementierenden Klassen.

ii. Erläutern Sie kurz den Unterschied zwischen einer virtuellen und einer rein virtuellen Methode.

Eine virtuelle Methode ist eine Methode, welche in der Basisklasse definiert ist und in der abgeleiteten Klasse überschrieben werden kann.
Eine rein-virtuelle Methode wird nur in der Basisklasse implementiert und muss in der abgeleiteten Klasse überschrieben werden
Darin liegt auch der Unterschied. Eine hat implementierung die andere nicht.

d) Von `Serialisation` soll eine Klasse `JSONSerialisation` abgeleitet werden, die eine JSON-Serialisierung implementiert. Dazu gibt es auch eine Ableitung der Parameter-Klasse, welche noch eine Konstante für den Namen der Wurzel hinzufügt:

```
01  class JSONParameter : public Parameter
02  {
03  private:
04    const char  rootnode[100];
05  public:
06    JSONParameter(const char* enc, int sz, const char* root);
07    const char* getRootnode() const;
08    // …
09  };
```

i. Welchen Fehler macht folgende Implementierung des allgemeinen Konstruktors `JSONParameter`? Wie könnte man diese beheben?

```
JSONParameter(const char* enc, int sz, const char* root) :
    Parameter(enc, sz), rootnode(root) {}
```

Das rootNode Feld wird direkt mit dem Wert root initialisiert. Da es ein Array ist geht das nicht ohne weiteres. Entweder man iteriert durch rootnode und fügt den jeweiligen Wert in root oder benutzt sowas wie strncpy(rootnode, root, 100). Man sollte auch prüfen, ob

die Werte & Größe zum Array passen.

ii. Welche Arten von Methoden außer Konstruktoren und Destruktoren können in C++ generell nicht überschrieben werden? Geben Sie mindestens 2 Möglichkeiten an und führen Sie die notwendige Syntax auf.

Final Methoden: final gekennzeichnete Methoden in der Basisklasse sind unveränderbar in der abgeleiteten Klasse.

static Methoden: nicht Objekt abhängig, sondern wird von der Klasse selbst aufgerufen. Daher ist es nicht möglich diese zu überschreiben

operator Methoden: kann nur in der Basisklasse überschrieben werden

```
class Basis {
public:
  virtual void method1() final;
  static void method2();
  operator int() const;
  bool operator=(const Basis& other) const;
};
```

```
class Abgeleitet : public Basis {
public: // Errors!
  virtual void method1() final;
  static void method2();
  operator int() const;
  bool operator=(const Basis& other) override;
};
```

e) Die Ableitung `JSONSerialisation` übernimmt die Parameter aus der abgeleiteten Klasse, verwaltet selbst aber nur einen Zeiger auf die Basisklasse:

```
01  class JSONSerialisation : public Serialisation
02  {
03  private:
04    std::ostream& out;
05  public:
06    JSONSerialisation(Parameter* p, std::ostream& o);
07    ~JSONSerialisation();
08    // …
09  };
```

Implementieren Sie den Konstruktor von `JSONSerialisation` so, dass er, falls er über den Zeiger `p` ein Objekt vom Typ `JSONParameter` erhält, aus diesem den Wert `rootnode` entnimmt und auf den Stream `out` ausgibt. Falls es ein anderer Zeiger ist, soll nur `{"std":` ausgegeben werden.

```
JSONSerialisation::JSONSerialisation(Parameter* p, std::ostream& o):Serialisation(p),out(o) {
    if (JSONParameter *jsonp = dynamic_cast<JSONParameter*>(p)) {
        out << jsonp -> getRootnode();
    } else {
        out << "\"std\"";
```

f) Es soll noch der Destruktor von `JSONSerialisation` betrachtet werden.

i. Warum benötigt die Klasse `JSONSerialisation` einen Destruktor?

std ostream& out ist dynamisch verwalteter Speicher, welcher nicht automatisch freigegeben wird. Um Speicherverschwendung zu vermeiden muss der Destruktor aufgerufen werden. Der Destruktor wird automatisch aufgerufen falls man das Objekt verlässt oder explizit mit delete gelöscht wird.

ii. Welche Änderungen an den oben genannten Klassen sind vorzunehmen, so dass dieser korrekt funktioniert? Erläutern Sie kurz, was ohne diese Änderungen passieren würde?

Es könnte sein das der Destruktor nicht aufgerufen wird, da möglicherweise nur die Referenz gelöscht wird o.ä.
Die Basisklasse, sowie alle derivate müssen ihren dynamisch allokierten Speicher löschen.

```
class Serialisation {
public:
  virtual ~Serialisation() {
    //release ressources
  };
};
```

```
class Parameter {
public:
  virtual ~Parameter
    //release ressources
  };
};
```

```
class JSONSerialisation {
public:
  virtual ~Parameter
    //release ressources
  };
};
```

oder

```
JSONSerialisation::~JSONSerialisation() {
  //release resources
}
```

Aufgabe 5: C++: STL (19 Punkte)

Eine Klasse `Node` verwalte einen Knoten einer Cloud-Infrastruktur. Für folgende Betrachtung sei nur die IP-Adresse (address) relevant, die der Einfachheit halber hier als Zeichenkette gespeichert wird.

```
using namespace std;
class Node
{
    public:
        string getAddress() const;
        // andere Methoden...
};
```

Für eine Applikation wird ein Vektor aus Objekten dieser Klasse gebildet. Wir wollen diesen Vektor nun nach einer Adresse durchsuchen und die Ergebnisse numerisch, d.h. alphabetisch sortieren. (Eine Beschreibung der zu verwendenden Klassen und Funktionen finden Sie unterhalb dieser Aufgabe.)

Funktor-Klassen: Haben immer eine Operator()-Methode (überladung) → Ermöglichen Code zu kapseln
Klassenaufruf als Argument für Funktionen, welche eine Funktion erwarten.
F-Klassen können auch Daten speichern & über den Lebenszyklus behalten (ermöglicht speichern der Zustände zwischen Aufrufen)

a) Erstellen Sie dazu eine **Funktor-Klasse** `AddressSorter`. Diese soll eine gesuchte Zeichenfolge als `string`-Objekt aufnehmen und als Member speichern. In ihrer Operator-Methode soll sie ein Node-Objekt darauf prüfen, ob im Rückgabewert von `getAddress()` die gesuchte Zeichenfolge enthalten ist. Die Klasse soll auch über ein privates Element verfügen, welches mitzählt, wie viele übergebene Objekte die Bedingung erfüllen, und dafür eine Getter-Methode bereitstellen.

```
class AdressSorter() {
private:
  string searchString;
  int count;
public:
  AdressSorter(String s) : searchString(s), count(0) { }
  bool operator()(const Node&n) {
       if (n.getAdress().find(searchString) != string::npos) {
          count++;
          return true;
        }
        return false;
   }
   int getCount() const {return count;}
};
```

b) Schreiben Sie eine Funktion

`int search_nodes(const vector<Node>& nd, const string& ad)`

die zunächst einen Ergebnisvektor von Knoten in derselben Größe wie `nd` anlegt und dann mittels der Funktion `copy_if` alle Elemente aus `nd` kopiert, welche die in `ad` angegebene Zeichenkette enthalten. Verwenden Sie dazu ein Objekt der Klasse `AddressSorter`.

Anschließend wird der Ergebnisvektor mittels `resize()` auf die eigentlich nötige Größe reduziert. Dazu können Sie die Anzahl der gefundenen Übereinstimmungen aus dem Objekt vom Typ `AddressSorter` verwenden.

Sortieren Sie schließlich die Ergebnismenge mittels der Funktion `sort`. Verwenden Sie als Vergleichsfunktion einen **Lambda-Ausdruck**, in dem die Betreffe der beiden Nachrichten (`getAddress()`) mittels des Operators < verglichen werden.

```
int search_nodes(const vector<Node>& nd, const string& ad) {
    vector<Node> result(nd.size());
    AdressSorter sorter(ad);
    auto it = copy_if(nd.begin(), nd.end(), result.begin(), sorter);
    result.resize(sorter.getCount());
    sort(result.begin(), result.end(), [](const Node&a, const Node&b) {
        return result;
    }
}
```

Vektor nd nach Adresse ad durchsuchen
Erstellt einen Ergebnisvektor von Knoten mit Größe nd und kopiert alle Elemente mit Hilfe

c) Welche Vorteile bietet eine Funktor-Klasse oder ein Lambda-Ausdruck gegenüber einem Funktionszeiger bei Algorithmen wie `copy_if` oder `sort`?

1. Lesbarkeit: Kontext direkt ersichtlich ohne Verweis auf seperate Funk.def.
2. Flexibilität: Jede benötigte Funktionalität an jedem Ort an dem sie benötigt wird. Vergleichs- oder Filtermethoden in einziger Methode/Funktion zu verwenden.
3. Performance: in einigen Fällen schneller, da automatisch inline-expandiert, somit keine Aufrufe.
4. Einfachheit: Einfacher zu erstellen & verwenden, da keine seperate Deklaration, sondern direkt Code. Außerdem Reduktion von Verwaltung & übersichtlicherer Code.
5. Capture-List: Lambdas haben Möglichkeit aus äußeren Kontext Variablen zu erfassen & nutzen.

Nachteil: Lambdas & Funktoren müssen erst gelernt & verstanden werden

Klasse string

Methode	**Erklärung**
`string()`	Konstruktor, der einen leeren String erzeugt.
`int find(const string& str)`	Findet die Position, an der die Zeichenfolge `str` zum ersten Mal im String vorkommt. Falls sie nicht enthalten ist, wird die Konstante `string::npos` zurückgeliefert.
`bool operator< (const string& lhs, const string& rhs)`	Befreundete Operatorfunktion außerhalb der Klasse, die zwei Strings lexikographisch vergleicht

Funktion copy_if

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
  OutputIterator copy_if (InputIterator first, InputIterator last,
                          OutputIterator result, UnaryPredicate pred);
```

Die Funktion `copy_if` kopiert alle Elemente von `first` bis `last` in den Container, der mit `result` beginnt, wobei nur Objekte kopiert werden, welche die Bedingung `pred` erfüllen. Der Ergebniscontainer muss vorher angelegt sein, am besten mit so vielen Elemente wie auch der Eingabecontainer enthält (Methode `size()`). Der Rückgabewert zeigt auf das Ende des Ergebniscontainers.

Funktion sort

```
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Die Funktion `sort` sortiert einen Container von `first` bis `last` und verwendet dazu die übergebene Vergleichsfunktion.

Klasse vector<T>

Methode	**Erklärung**
`vector<T>()`	Konstruktor, der einen leeren Vektor erzeugt.
`void push_back(const T& value)`	Der Wert `value` wird an das Ende des Vektors gehängt.
`int size()`	Ermittelt die Anzahl der Elemente des Vektors.
`vector<T>::iterator begin()`	Liefert einen Iterator, der auf das erste Element verweist.
`vector<T>::iterator end()`	Liefert einen Iterator, der hinter das letzte Element verweist.
`void resize(unsigned n)`	Verändert die Größe des Vektors auf `n` Elemente

Viel Erfolg !

Anhang: Priorität von Operatoren in C

Priorität	Operatoren		Assoziativität
1	**()**	Funktionsaufruf	links
	[]	Array-Index	links
	-> .	Memberzugriff	links
2	**! ~**	Negation (logisch, bitweise)	rechts
	++ --	Innkrement, Dekrement	rechts
	sizeof	Sizeof-Operator	rechts
	+ -	Vorzeichen (unär)	rechts
	(Typname)	cast	rechts
	*** &**	Dereferenzierung, Adresse	rechts
3	*** /**	Multiplikation, Division	links
	%	modulo	links
4	**+ -**	Summe, Differenz (binär)	links
5	**<< >>**	bitweises Schieben	links
6	**< <=**	Vergleich kleiner, kleiner gleich	links
	> >=	Vergleich größer, größer gleich	links
7	**== !=**	Gleichheit, Ungleichheit	links
8	**&**	bitweises AND	links
9	**^**	bitweises XOR	links
10	**|**	bitweises OR	links
11	**&&**	logisches AND	links
12	**||**	logisches OR	links
13	**?:**	bedingte Auswertung	rechts
14	**=**	Wertzuweisung	rechts
	+= -= ***= /=** **%= &=** **^= |=** **<<= >>=**	kombinierter Zuweisungsoperator	rechts
15	**,**	Komma-Operator	links

"C als erste Programmiersprache (4.Auflage)" S.552