

## BACHELORPRÜFUNG

Prüfungsfach: **Fortgeschrittene Programmierung**  
Prüfungstag: 30. Juli 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	Datentyp
11//3/2	1.5	float
2e+1j	20j	complex
[1,2] + [(3,4)]	[1, 2, (3, 4)]	list
(3, 5, 7, 9)[::-1]	(9, 7, 5, 3)	tuple

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

- a) Definieren Sie eine Liste von Tripel (3er Tupeln), die alle Möglichkeiten darstellt, wie man drei Kugeln mit 1, 2 und 3 beschriftet aus einer Urne ziehen kann.

$[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)] \rightarrow$  ohne Wdh.

$\text{triplets} = [(i, j, k) \text{ for } i \text{ in } [1, 2, 3] \text{ for } j \text{ in } [1, 2, 3] \text{ for } k \text{ in } [1, 2, 3]] \rightarrow$  mit Wdh.

- b) Schreiben Sie einen lambda-Ausdruck, der eine zweistellige Funktion (mit zwei Parametern x und y) darstellt, deren Rückgabewert eine Liste aller geraden Zahlen zwischen x und y ist. Damit soll folgende Ausgabe erzeugt werden:

```
>>> z = list(map(lambda  , [1, 3], [7, 9]))
>>> print(z)
[[2, 4, 6], [4, 6, 8]]
```

$\text{lambda } x, y: [i \text{ for } i \text{ in range}(x, y) \text{ if not } i \% 2]$

### Aufgabe 3: Python: Klassen (12 Punkte)

Schreiben Sie eine Python-Klasse `Messwert`. Diese soll folgende Elemente enthalten:

- a) Eine private Variable `wert` vom Typ `float`
- b) Einen allgemeinen Konstruktor, mit dem der Wert mit einem übergebenen Parameter initialisiert werden kann
- c) Eine Getter- und eine Setter-Methode für den Wert als Property.
- d) Eine Methode `__str__()` für Print-Ausgaben.
- e) Ein Testskript, das alle Methoden der Klasse nutzt und mit sinnvollen Werten aufruft.

```
class Messwert:
    a)  __wert : float
    b)  __init__(self, wert):
        self.__wert = wert
    c)  @property  -----> einfache Möglichkeit eine Methode als
        def get_wert(self):                               Eigenschaft zu deklarieren.
            return self.__wert
        @setter-methode
        def set_wert(self, auto a):
            self.__wert = a
    d)  def __str__(self):
        print(self.__wert)
    e)
        Messwert m1 (69.42)
        Messwert m2 (7.13)
        w = m2.get_wert()
        m1.set_wert(w)
        m1.__str__()
```

#### Aufgabe 4: C++: Klassen, Konstruktoren und Destruktoren, Vererbung (31 Punkte)

Mit Hilfe des Entwurfsmusters „Strategie“ soll eine Möglichkeit geschaffen werden, Byte-arrays, d.h. Zeichenketten oder Binärdaten, mit unterschiedlichen Verfahren zu verschlüsseln. Die Verfahren verfügen dabei über verschiedene Systemparameter (neben Schlüssel z.B. die Bitbreite), die ebenfalls in Klassen repräsentiert werden sollen.

```
class Encryption
{
private:
    Parameter* params;
public:
    Encryption(Parameter* p = nullptr);
    ~Encryption();
    virtual int encrypt(const char* source, char* dest, int size);
    virtual int decrypt(const char* source, char* dest, int size);
};

class Parameter
{
private:
    char* key;
    unsigned int keysize;
public:
    Parameter();
    Parameter(const char* k, unsigned int ksize);
    Parameter(const Parameter& p);
    ~Parameter();
    // ...
};
```

- a) Die Klasse `Encryption` soll die Schnittstelle für die verschiedenen Verschlüsselungsverfahren werden. Wie ist die Klasse zu ändern, damit die Methoden `encrypt` und `decrypt` rein virtuell sind? Welche Auswirkungen hat das?

```
virtual int encrypt(const char* source, char* dest, int size) = 0;
virtual int decrypt(const char* source, char* dest, int size) = 0;
```

Damit keine Objekte erschaffen werden können,  
welche nicht direkt initialisiert & damit verwendet  
werden. → abstrakte Klasse

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

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

Da dynamische Speicherverwaltung vorhanden. Falls das Objekt der Klasse `Parameter` einem anderen Objekt zugewiesen wird, dann muss der Speicher der Variablen freigegeben werden, da sonst ein Speicherleck entstehen kann. `char* key` ist der dynamisch allozierter Speicher

- ii. Geben Sie eine Implementierung des Zuweisungsoperators nach dem **copy&swap-Idiom** an.

```
Parameter& operator = (const Parameter& other) {  
    if (this != &other) {  
        Parameter tmp(other);  
        std::swap(key, tmp.key);  
        std::swap(keysize, tmp.keysize);  
    }  
    return *this  
}
```

c) 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. Warum ist es für die Klasse `Parameter` sinnvoll, einen zu schreiben?

Da wahrscheinlich Ressourcen z.B. dynamisch allozierter Speicher für die Schlüssel (`key`) verwaltet wird, welche bei der Übertragung unangekündigt bleiben können. Das verbessert die Leistung, da nicht unnötig kopiert wird.

- iii. Können auch von `Parameter` abgeleitete Klassen diesen Operator nutzen? (Kurze Begründung!)

Die abgeleitete Klasse erbt die Methode, da diese in der Basisklasse definiert wurde. Solange kein weiterer dynamisch allozierter Speicher dazu kommt wird alles richtig bearbeitet, ansonsten muss die abgeleitete Klasse den Operator überschreiben.

- iv. Schreiben Sie eine Implementierung für diesen Operator.

```
Parameter& operator=(const Parameter&& other) {  
    std::swap(key, other.key);  
    std::swap(keysize, other.keysize);  
    return *this;  
}
```

- d) Von Encryption soll eine Klasse `DESEncryption` abgeleitet werden, die das Verfahren DES implementiert. Dazu gibt es auch eine Ableitung der `Parameter`-Klasse:

```
class DESParameter : public Parameter  
{  
private:  
    int    bitsize;  
public:  
    DESParameter(const char* k, int sz, int bits);  
    int getBitsize() const;  
    // ...  
};
```

Implementieren Sie den Konstruktor `DESParameter`.

```
DESParameter::DESParameter(const char* k, int sz, int bits): Parameter(k, sz), bitsize(bits) {}
```

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

```
class DESEncryption : public Encryption
{
private:
    int bitsize;
public:
    DESEncryption(Parameter* p);
    ~DESEncryption();
    // ...
};
```

Implementieren Sie den Konstruktor von `DESEncryption` so, dass er, falls er über den Zeiger `p` ein Objekt vom Typ `DESParameter` erhält, aus diesem den Wert `bitsize` entnimmt und in der privaten Variable der Klasse speichert. Falls es ein anderer Zeiger ist, soll `bitsize` auf den Wert -1 gesetzt werden. Erläutern Sie kurz Ihre Vorgehensweise.

```
DESEncryption::DESEncryption(Parameter* p): Encryption(p) {
    DESParameter* des_p = dynamic_cast<DESParameter*>(p);
    if (des_p) {
        bitsize = des_p->getBitsize();
    } else {
        bitsize = -1;
    }
}
```

### Aufgabe 5: C++: Operatorüberladung (13 P.)

Betrachten Sie folgende Klasse, die einen „intelligenten“ Zeiger repräsentieren soll:

```
template<typename T>
class Pointer
{
private:
    T* p;

public:
    Pointer(T* arg = nullptr) : p(arg) {};
    ~Pointer() {
        if (p != nullptr) delete p;
    }
};
```

Erweitern Sie diese Klasse um folgende Methoden (zu definieren außerhalb der Klassendeclaration):

a) Die Operatoren \* und -> für die Dereferenzierung.

```
T& operator* {return *p};
T* operator-> {return p};
```

b) Einen Inkrementoperator für Postfix-Notation.

```
Pointer& operator++(int) {
    p++;
    return *this;
}

Pointer& operator++() {
    ++p;
    return *this;
}
```

Postfix-Notation

Prefix-Notation

c) Einen Operator, der folgenden Zugriff erlaubt:

```
Pointer<int> ptr;  
// ... ptr füllen  
  
if (ptr) {  
    // ... tu etwas  
}  
  
operator bool() const {  
    return p != nullptr;  
}
```

#### Aufgabe 6: C++: STL (16 Punkte)

a) Zum Bearbeiten von STL-Containern gibt es verschiedene Datenstrukturen, u.a. den Iterator. Was ist ein Iterator? Warum verwendet man gerne Iteratoren zum Zugriff auf solche Datenstrukturen?

Eine Art Zeiger, der auf ein Element innerhalb der Datenstruktur verweist & es ermöglicht, die Elemente sequenziell zu durchlaufen, ohne die Struktur zu kennen.  
Iteratoren werden gerne verwendet, weil sie eine abstrakte Sicht auf die Datenstruktur bieten & somit Implementierungsdetails verborgen sind. Auf diese Weise kann man Code schreiben, der unabhängig von der tatsächlichen Datenstruktur ist & somit leichter wiederverwendbar & wartbar ist.  
Außerdem ermöglicht es auf einheitliche Art & Weise auf verschiedene Datenstrukturen zuzugreifen & diese zu durchlaufen.

b) Welche Operatoren muss ein Iterator mindestens unterstützen? Nennen Sie mindestens drei (Syntax und Beschreibung).

Ein Iterator muss mindestens die Operatoren `*`, `++` & `!=` unterstützen.  
`*`: Gibt den Wert des Elements zurück, auf das der Iterator zeigt  
`++`: Inkrementiert den Iterator, sodass es auf das nächste Element verweist  
`!=`: Prüft ob Iterator auf das gleiche Element wie ein anderer Iterator verweist.  
Diese sind Pflicht.  
Optionale: `--`, `+`, `-`, `<`, `>`, `<=`, `>=`



c) Wir betrachten einen STL-Vektor von Ganzzahlen. Schreiben Sie eine Funktion

```
std::vector<int> search_even(const std::vector<int>& numbers);
```

welche die übergebene Liste von Ganzzahlen durchläuft und alle geraden Zahlen in einen Ergebnisvektor übernimmt, den die Funktion dann zurückliefert. Gehen Sie dabei wie folgt vor:

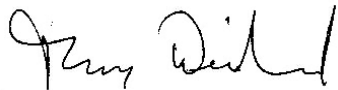
- i) Definieren Sie einen Lambda-Ausdruck, der bestimmt, ob eine Ganzzahl gerade ist, und speichern Sie diesen in einer lokalen Variablen namens `even`.
- ii) Durchlaufen Sie in einer Schleife den übergebenen Vektor mittels eines Iterators bis zum Ende.
- iii) Prüfen Sie in dieser Schleife durch Aufruf der Standard-Funktion `find_if()`, ob sich eine weitere gerade Zahl im Vektor befindet. Dabei erwartet `find_if` drei Argumente: Anfang und Ende des zu durchsuchenden Bereichs als Iterator sowie eine Entscheidungsfunktion. Hierfür können Sie Ihren Ausdruck `even` verwenden. Als Ergebnis liefert `find_if` einen Iterator auf das gefundene Element oder auf das Ende des Vektors zurück.
- iv) Sobald Sie eine Zahl gefunden haben, fügen Sie diese in den Ergebnisvektor über die Methode `push_back()` ein.
- v) Die Funktion liefert am Ende den Ergebnisvektor zurück.

```
std::vector<int> search_even(const std::vector<int>& numbers) {  
    std::vector<int> Ergebnis  
    i) auto even = [](int number) { return number % 2 == 0; };  
    ii) for(auto it = numbers.begin(); it != numbers.end(); ++it) {  
    iii)         auto found = std::find_if(it, numbers.end(), even);  
                if(found != numbers.end()) {  
    iv)             Ergebnis.push_back(*found);  
                }  
    }  
    v) return Ergebnis  
}
```

### Klasse template `vector<typename T>`

Methode	Erklärung
<code>vector&lt;T&gt;()</code>	Konstruktor, der einen leeren Vektor erzeugt.
<code>void push_back(const type&amp; value)</code> <i>*p</i>	Der Wert value wird an das Ende des Vektors gehängt.
<code>int size()</code>	Ermittelt die Länge der Liste.
<code>vector&lt;T&gt;::iterator begin()</code>	Liefert einen Iterator, der auf das erste Element verweist.
<code>vector&lt;T&gt;::iterator end()</code>	Liefert einen Iterator, der hinter das letzte Element verweist.

Viel Erfolg!



# Anhang: Priorität von Operatoren in C

Priorität	Operatoren		Assoziativität
1	()	Funktionsaufruf	links
	[]	Array-Index	links
	-> .	Memberzugriff	links
2	! ~	Negation (logisch, bitweise)	rechts
	++ --	Inkrement, 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