Vorlesungsprüfung Programmiersprache 1

Termin der Prüfungen:

1. Prüfungstermin: Do 17.03.2022 16:00-18:00
2. Prüfungstermin: Do 12.05.2022 16:00-17:00
3. Prüfungstermin: Di 14.06.2022 16:00-17:00

Prüfungsablauf:

Die 1. Prüfung erfolgt in mehreren Gruppen, wahrscheinlich jede halbe Stunde startend. Die Gruppeneinteilung wird noch von mir veröffentlicht. Die individuellen Prüfungsfragen versende ich per Email als PDF-Dateien beim Start der Gruppen. Die Teilnehmer senden mir darauf die Prüfungsantworten in Form einer ausgefüllten Antwortdatei (als Attachment!) als Antwort auf meine Mail zurück. Die Vorlage für die Antwortdatei finden Sie in der Datei **abgabe_muster.txt** auf meiner Homepage. Laden Sie diese Datei herunter, füllen Sie diese aus und senden Sie sie als Mailanhang an mich zurück.

Wie ist die Antwortdatei auszufüllen: Die Angabenummer finden Sie in ihrer Prüfungsdatei. Ohne diese Angabenummer kann ich Ihre Prüfung nicht korrigieren! Falls Sie z.B. bei Frage 7 nur die Teilantworten b und c für richtig halten, füllen Sie die entsprechende Zeile so aus:

7 bc

Falls Sie alle Teilantworten a-d für richtig halten, schreiben Sie:

7 abcd

Falls Sie keinen der möglichen Unterpunkte für richtig halten, belassen Sie es bei

7

Bitte schreiben Sie die Kleinbuchstaben **ohne Trennzeichen oder Leerzeichen** hintereinander!

Prüfungsstoff:

Alle Prüfungsteilnehmer, die **NICHT** Python in ihrer Vorlesung gehabt haben (alle vor WS 2020/21), bekommen die Prüfungsfragen über den damaligen alten Vorlesungsstoff (nur C++). Alle Teilnehmer der aktuellen Vorlesungen erhalten Prüfungsfragen zu Python **und** C++. Die folgende Aufstellung bezieht sich auf den Prüfungsstoff zur Vorlesung ab WS 2020/21.

Prüfungsstoff Python:

Der Prüfungsstoff umfasst den Stoff der Vorlesungsstunden 1-3:

Datentypen (int, float, bool, str) und Operationen (+,-,*,/,%), = und ==
Funktionen
if-Blöcke
Schleifen und continue, break, range(…)
Listen und ihre Methoden (z.B. append, count) und Teillisten, Indices, Länge
Tupel und Teiltupel, Indices, Länge
List-Comprehension und Summen

Es werden keine Fragen zu Bibliotheksmodulen gestellt (weder math, random, matplotlib) und auch nicht zu Dateien

Prüfungsstoff C++:

Der Prüfungsstoff umfasst die Vorlesungen 4-12, d.h. es werden keine Fragen zu Multithreading, Vererbung, Exceptions, Kommandozeile gestellt.

1) Ganzzahltypen: `char, short (int), int, long (int), unsigned char`
   `char <= short <= int <= long`, aber keine genaue Festlegung des Zahlenbereichs
   1 Byte: [-128, 127], bzw. bei unsigned [0,255]
   2 Byte: [-32000,32000], bzw. [0, 65000]
   4 Byte: [-2Mrd, 2Mrd], bzw. [0, 4Mrd]
   8 Byte: viel größer
   Konstanten wie 1L (1 vom Type long int), 1U (1 vom Typ unsigned int)
   Es ist nicht festgelegt, ob `char` auch negative Werte haben kann.
   `signed char, short, int, long, long long` haben positive UND negative Werte, alle `unsigned` Typen nur positive oder 0.

2) Ganzzahloperatoren +,-,*,/,%, ++, --: Problematik des Overflows
   Berechnungen wie : `int i = 4; (i++)%3 = ?`
   mehrere Operationen werden implizit geklammert , Assoziativität von links
   Ergebnistyp leitet sich aus den Typen der Operanden ab (ist oft der „größere“ Typ der Operanden, aber mindestens int) `4/3 -> 1; 4/3 + 1.0 -> 2.0;`
   Überlauf: Alle Ganzzahltypen können überlaufen, bei den **signed-Typen** ist das **undefined Behaviour** (d.h. es kann in einem solchen Fall alles passieren, von Abbruch bis zum Absturz!), bei **unsigned Typen** werden die überzähligen höchstwertigen Bits weggeschnitten und es wird normal weitergerechnet (es ist defined Behaviour)
   z.B. bei 4 Byte int: 2* 2 Mrd < 0 (eigentlich undefined Behaviour), in der Praxis fast immer Überlauf!
3) Gleitkomma-Typen: `float` (7 Stellen), `double` (15 Stellen); Rundungsfehler:
   `if (x == y)` (schlecht, besser `if (std::abs(x-y) < 1e-14)`
   `for (double x = 0.1; x <= 1.0; x += 0.1)` schlecht, besser:
   `for (double x = 0.1; x <= 1.05; x += 0.1)`
   Operatoren +,-,*,/
   Konstante haben Dezimalpunkt und/oder Exponenten: `1e0 = 1.0` (double!!)

```
int i = 0,  j = 2;
double x = ++i/j--;
```

   Werte von `i, j, x` nach diesem Code?
4) logische Operatoren: !, ||, &&, < ,<=,>,>=,==,!=: Ergebnis `false` (0) oder `true` (1)
5) Bit-Operatoren: ~, &, |, ^, <<, >> (keine Fragen dazu außer ^ ist keine Potenz!)
6) Zuweisungsoperatoren: `=, +=, −=` usw. Auf der linken Seite muss immer veränderbarer Ausdruck stehen (nichtkonstante Variable etc.). Der Rückgabewert des Zuweisungsoperators ist bei den Standard-Typen der zugewiesene Wert:
   `int i;  i = 3*j;`
   `int i = 1;` **DAS IST KEINE ZUWEISUNG!**
7) C-Arrays: Diese wurden in der LV durch den `std::array` Container ersetzt und Fragen beziehen sich nur auf dieses C++-Typtemplate: `std::array<int,10> x;` x umfasst 10 int, Index 0 - 9: Indexüberlauf ist unbedingt zu vermeiden; Initialisierung mit {} erlaubt (x = {0, 1}; oder auch x{0, 1}; fehlende Werte werden mit Nullen ergänzt);
8) Container und Algorithmen:
   beachte den Unterschied zwischen {} und () bei den STL-Containern:
   `std::vector<double> a{200, 2}` hat 2 Elemente 200, 2
   `std::vector<double> a(200, 2)` hat 200 Elemente mit Wert 2

   `begin()` und `end()` sind Iteratoren, also keine Container-Elemente und (meist) auch keine Pointer (obwohl sie dereferenziert werden können!):
   `a.begin() = 5;` Fehler!!
   `*a.begin() = 5;` ok (besser ist: `a.front() = 0;`)

9) Schleifen, Bedingungen: Nur 1 Anweisung: oder {}, continue, break-Anweisung

```
int i = 1;
```

```
if   (i = 5)
      i++;
```

Wert von i nach diesem Code?

```
int i = 1;
if   (i == 5);
      i++;
```

Wert von i nach diesem Code?

10) Überladen von Funktionen: Wann dürfen Funktionen denselben Namen haben:
    - in C und Python gar nicht
    - in C++: Wenn sie sich in der Argumentliste unterscheiden (NICHT: Rückgabetyp!)
    - in C++: Wenn sie in verschiedenen Namespaces liegen (z.B. auch Klassenmethoden von verschiedenen Klassen)

    Auch wenn die Definition von überladenen Funktionen erlaubt ist, kann beim Aufruf ein Fehler passieren (wenn z.B. die Funktion nicht eindeutig bestimmbar ist, da die Argumenttypen nicht genau passen und es dann mehrere zulässige Aufrufe gibt
11) `auto` als Rückgabetyp bei Funktionen und Funktionstemplates ist seit C++11 (eingeschränkt) und seit C++14 (uneingeschränkt) erlaubt. Vor C++20 durfte `auto` nicht als Typ von Funktions-Argumenten oder Funktionstemplate-Argumenten verwendet werden. Erst ab C++20 ist das möglich und erzeugt immer ein Funktionstemplate (neue vereinfachte Templateform).
12) Lambdas (anonyme Funktionen) `auto f = [](auto x) { return x*x; };` Seit C++11 erlaubt, seit C++14 darf `auto` der Typ der Argumente sein. Alle Übergabemechanismen sind erlaubt.
13) Übergabe per Referenz: `void f(int& j)` (`j` wird per Referenz übergeben: Original **kann** verändert werden). *`void g(int j)`* (`j` wird per Wert = als Kopie übergeben: Original **kann nicht** verändert werden), Übergabe per const-Referenz: `void h(const int& i) {}` (Argument **darf nicht** verändert werden, weder direkt in der Funktion noch indirekt durch Aufrufe anderer Funktionen!)
14) Objekte: siehe Vorlesungstext. `class` und `struct`: enthalten Attribute (Variablen) und Methoden, diese können `private`: oder `public`: (oder `protected`:) sein. Die Defaulteinstellung bei bei `class` ist `private`:, bei `struct public`: Methoden, die das Referenzobjekt nicht ändern, sollten unbedingt als `const` deklariert werden (konstante Methoden)

    Konstruktor (Methode mit Name gleich wie Klassenname, OHNE Typ), Destruktor (Methode mit Name gleich wie ~Klassennname, OHNE Typ, OHNE Parameter)

    Konstruktoren rufen (vor dem Funktionsrumpf in {}) für ALLE Basisklassen und ALLE Attribute, die selbst echte Objekte sind, deren Konstruktoren auf (in der Reihenfolge, wie diese definiert werden)
    Konstruktoren können/sollen Initialisierungslisten enthalten: Diese können für alle Basisklassen und alle Attribute die aufzurufenden Konstruktoren festlegen

    Destruktoren rufen (nach ihrem Code) für alle Attribute, die echte Objekte sind, und für die Bsaisklassen deren Destruktor auf.

In einer Objektmethode zeigt der Pointer `this` (der ist automatisch definiert) auf das Referenzobjekt. Auf dessen Teile oder dessen Methoden kann **innerhalb** von Objektmethoden auch ohne `this->` zugegriffen werden.

15) Konstruktoren und Zuweisungsoperatoren:
Standardkonstruktor = default Constructor: Konstruktor der ohne Argumente aufgerufen werden kann (weil er keine hat oder weil für alle Argumente ein Default-Wert definiert ist)!
C++ erstellt automatisch einen trivialen Defaultkonstruktor, wenn die Klasse KEINEN User-definierten Konstruktor besitzt => jede Klasse hat mindestens einen Konstruktor (aber nicht unbedingt einen Standard-Konstruktor)

```
class/struct  A {...};   // definiere ein echtes Objekt (kein POD)
A x;          // hier wird der Standard-Konstruktor aufgerufen, Fehler: wenn keiner existiert
              // oder nicht aufgerufen werden darf (z.B. da nicht public:)
A  x{5};      // hier wird der Konstruktor aufgerufen, der mit einem int-Argument
              // aufgerufen  werden kann
A  x = 5;     // entspricht A  x{5};  (KEINE Zuweisung, sondern Konstruktor!)
```

Copy-Konstruktor: Initialisiert ein Objekt als Kopie eines anderen Objekts desselben Typs:

```
A y{x};     //  y wird als Kopie von x  erzeugt;
A y = x;    //  dasselbe
A z; z = x;       // Zuweisungsoperator
```

16) Destruktor: Wird automatisch beim Vernichten des Klassenobjekts aufgerufen. Das geschieht am Ende des Programmblocks, in dem das Objekt definiert wurde.
17) Wann erzeugt C++ Default-Konstruktor, Destruktor, Copy Constructor, Zuweisungsoperator:
Die Regeln sind seit C++11 wesentlich komplexer geworden. Ich bin mit folgenden Regeln zufrieden: C++ erzeugt diese Dinge, wenn sie benötigt werden UND wenn:
(Default-Konstruktor:) das Objekt KEINEN userdefinierten Konstruktor besitzt
(Destruktor:) das Objekt KEINEN userdefinierten Destruktor besitzt
(Copy-Konstruktor:) das Objekt KEINEN Userdefinierten Copy-Konstruktor besitzt
(Zuweisungsoperator:) das Objekt KEINEN Userdefinierten Zuweisungsoperator besitzt

Infolge meiner vereinfachten Regeln gilt (für die Objekte in den Prüfungsfragen):
Jedes Objekt hat einen Copy-Konstruktor (entweder userdefiniert oder C++ erzeugt)
Jedes Objekt hat einen Zuweisungsoperator (entweder userdefiniert oder C++ erzeugt)
Jedes Objekt hat einen Destruktor (entweder userdefiniert oder C++ erzeugt)
Nicht jedes Objekt hat einen Default-Konstruktor
Jedes Objekt hat einen Konstruktor
Das heißt noch lange nicht, dass diese Methoden alle `public`: sind!

18) Konstruktoren, die mit genau einem Argument eines anderen Typs aufrufbar sind und die nicht `explicit` sind, werden als von C++ automatisch als Umwandlungs-Operatoren verwendet.

19) Operatoren: `operator+()` etc. können überladen werden (für eigene Klassen)
z.B. `class Bruch`

20) `iostream`:
`std::cin, std::cout, std::cerr` (nach `using namespace std;` darf man auch `cin, cout, cerr` schreiben)
erfordert: `#include <iostream>`
Ausgaben könne verkettet werden: `cout << x << y;` (der Rückgabewert von `operator<<()` muss dazu der Stream selbst sein)
die Pfeile << oder >> weisen immer in die Richtung des Datenstroms

`std::cin >> x;`
Steht das Konstrukt in einem `if, for, while` etc. wird der Erfolg der Eingabe getestet:

```
if (cin >> i)
     cout << "Eingabe war erfolgreich\n";
while (cin >> x)      // Solange das Einlesen von x erfolgreich ist
{ … }
```

Stream-Eingaben müssen das Objekt verändern, brauchen daher das Objekt als Referenz!

21) File-Streams: `#include <fstream>` (und zusätzlich `#include <iostream>`)
EingabeStreams (`std::ifstream`) und Ausgabestreams (`std::ofstream`) werden am besten per Konstruktor geöffnet (oder angelegt) und durch den Destruktor automatisch geschlossen:

```
ifstream  in{"Input.dat"};     // öffne Datei "Input.dat" als stream in
if (!in.is_open())             // Öffnen hat nicht geklappt!
     usage("Kann Eingabedatei Input.dat nicht lesen");
else
     in >> i;                  // lies mittels >> usw.

ofstream  out("Ouput.dat");    // öffne Datei "Output.dat" bzw. lege sie an;
                               //  sie ist jetzt SICHER LEER!!!!
if (! out.is_open())           // Öffnen hat nicht geklappt!
     usage("Kann Ausgabedatei Output.dat nicht schreiben");
else
     out << i;                 // schreib mittels << usw.
```

Testbeispiele für Objekte:

```
class A1 {
     int i;
     A1(int k = 1234) : i{k} {}
};

class A2 {
     int i;
public:
     A2(int k = 1234) : i{k} {}
};

class B {
     int i;
public:
     explicit B(int k) : i{k}  {}
};

struct C {
public:
     std::string s;
};

class D {
     int i;
};

int main()
{    A1 a1;     // Fehler!
     A2 a2;     // korrekt
     B  b;      // Fehler!
     B  b{4};   // korrekt!
     C  c;      // korrekt
     D  d;      // korrekt
```

Welche haben Default-Konstruktur:

`A1, A2`: ja, User-defined (aber bei `A1` `private:`)

`B`: nein, da ein User-definierter Konstruktor vorhanden ist.
`C`: ja, C++ defined
`D`: ja, C++ defined

Wann definiert C++ Default-Konstruktor:
Wenn einer benötigt wird und gar kein User-definierter Konstruktor vorhanden ist.

```
a2 = 5;    // korrekt: 5 wird durch den Konstruktor in ein A gewandelt,
           // der Zuweisungsoperator ist C++ erzeugt!
b = 3;     // Fehler!  3 wird durch Konstruktor nicht  in ein B gewandelt, da explicit
b = {3};   // Fehler! Konstruktor ist  explicit
b = B{3}; // korrekt!
c = "Ein Text";      // Fehler!
          // Der C-String kann nicht in den Typ  C  gewandelt werden
c.s = "Ein Text"; // korrekt (s ist public, C-String wird in C++ string gewandelt)
d = 1;    // Fehler:  Keine Umwandlung (int -> D) definiert
d = {1}; // korrekt, {1} wird zu einem D-Objekt gewandelt und zugewiesen!
d.i = 1; // korrekt
```