

Ü1

Übungsblatt: Von CP zu Eiffel und C*

Anforderungen

Das *Informatik 3 Praktikum* fordert von Ihnen selbstständigeres Arbeiten als die Praktika zu *Informatik 1* und *2*. Insbesondere arbeiten Sie sich selbstständig in die Entwicklungsumgebungen ein.

Zum Bestehen des Praktikums müssen Sie die Übungsaufgaben lösen. Sie geben Ihre Ergebnisse nicht ab, sondern legen sie dem Dozenten zur Kontrolle vor, falls bei der Aufgabe nichts anderes angegeben ist.

Sie dürfen sich gerne in größeren Gruppen gegenseitig beim Einarbeiten in die Entwicklungsumgebungen helfen sowie Problem- und Entwurfsdiskussionen zu den Aufgaben führen. Implementationen zu Programmieraufgaben können Sie wie gewohnt in Zweiergruppen bearbeiten. Der Anteil jedes Partners an der Lösung muss erkennbar sein. Papier- und Rechercheaufgaben lösen Sie besser einzeln.

Vier Implementations-sprachen

Die Programmieraufgaben sind in der Regel in den Implementationssprachen *Eiffel*, *C++*, *Java*, *C#* zu implementieren. Die empfohlene Reihenfolge ist die genannte, bei den C*-Sprachen ist eine andere Reihenfolge möglich. Durch mehrfaches Implementieren einer Aufgabe in verschiedenen Sprachen erkennen Sie, dass diese Sprachen auf gemeinsamen Konzepten basieren, und das Wesentliche an einer guten Implementation ein guter Entwurf ist, demgegenüber verschiedene Realisierungen von Sprachkonzepten nachrangig (nicht irrelevant) sind.

Aufgabe Ü1.1 Linux- Benutzeroberfläche

Machen Sie sich mit der Benutzeroberfläche des *Linux*-Betriebssystems vertraut! Lernen Sie die wichtigsten Kommandos der *Shell*! Falls Sie *Linux* auf Ihrem eigenen Rechner nutzen, sollten Sie Ihren KommilitonInnen beim Einarbeiten helfen.

Aufgabe Ü1.2 Hello World

Schreiben Sie in jeder der vier Sprachen ein **Hello-World-Programm**, um die Entwicklungsumgebung kennen zu lernen! Erweitern Sie Ihre Lösung spielerisch um Elemente Ihrer Wahl!

Bemerkung zu folgenden Aufgaben

Die folgenden Aufgaben wurden im *Informatik 2 Praktikum* im Sommersemester 2004 gestellt, aber selten bearbeitet. Sie sollen die Musterlösungen von *Component Pascal* (CP) in die vier anderen Sprachen manuell übertragen. Bekannte Lösungen zu reimplementieren ist ein schneller Weg, um die Syntax und Semantik einer neuen Sprache kennen zu lernen. Der Vorteil ist, dass Sie nicht gleichzeitig ein neues Problem lösen *und* eine neue Sprache lernen müssen. Außerdem lernen Sie auch beim Lesen und Verstehen guter Lösungen. Für das Vorgehen beachten Sie folgende **Richtlinien**:

Richtlinien

- (1) **Entwürfe übernehmen:** Sie behalten die Datenstrukturen und Algorithmen der Musterlösungen bei, denn nur so lassen sich die Sprachen vergleichen. Teilweise führen Sie Messungen durch, um die Laufzeit der erzeugten Objektcodes zu vergleichen, was nur mit gleichen Entwürfen sinnvoll ist. Das heißt z.B., dass Sie eine rekursive Lösung *nicht* durch eine iterative ersetzen, einen Top-Down- *nicht* durch einen Bottom-Up-Parser, da deren Laufzeitverhalten verschieden ist.
- (2) **Effizient implementieren:** Zunächst nutzen Sie die Sprachkonstrukte, die denen von CP am nächsten entsprechen und die Sie zuerst kennen. Wenn Sie die Zielsprache besser kennen, experimentieren Sie mit Lösungsvarianten und wählen effizientere Sprachkonstrukte. Das heißt z.B.: CP kennt keine Reihungskonstan-

ten, es muss Reihungen mit Anweisungen initialisieren. Sie übertragen das zunächst direkt in entsprechende C++-Anweisungen. Wenn Sie später mehr über C++ wissen, implementieren Sie die konstante Reihung effizienter mit einem Template. Sie wählen also jeweils die effizienteste Implementation, ohne das Wesen des Lösungsentwurfs zu ändern.

Aufgabe Ü1.3

Web-Recherche:
Design by Contract für C*

Eiffel unterstützt *Design by Contract (DbC)*. Wie sieht es bei den C*-Sprachen aus? D, das C++, Java, C# mit anderen bekannten Konzepten kombiniert, bietet auch DbC-Konstrukte. In C++ können Sie das `assert()`-Makro oder besser das `Assert()`-Template nutzen. Aber in Java und C# fehlen sogar Zusicherungen.

- (1) Erkunden Sie mit einer **Web-Recherche**, welche Werkzeuge es gibt, die **DbC für C*-Sprachen** unterstützen! Stellen Sie gute Links zusammen, beschreiben Sie jeden Link kurz und senden Sie die kommentierte Link-Liste per E-Mail an karlheinz.hug@fh-reutlingen.de!
- (2) Wählen Sie eine kostenlose Lösung, die **einfaches DbC** etwa wie mit Zusicherungen in CP mit wenigen Klassen erlaubt, laden Sie diese Klassen herunter und setzen Sie sie in Ihren Programmen ein!

Aufgabe Ü1.4

Prozedurales
Programmieren:
Edle Zahlen

Zweck dieser Aufgabe ist, die Sprachkonstrukte zur **Ablaufkontrolle** und **stromorientierte Ausgabe** kennen zu lernen und im **prozeduralen Stil** mit **Rekursion** zu programmieren. Dazu wählen wir *I2P SS04 Ü2 A3: Edle Zahlen*.

- (1) Studieren Sie die Musterlösung und übertragen Sie sie in die vier Sprachen!
- (2) Führen Sie **Laufzeitmessungen** durch! Das in CP benutzte Messmodul Utilities-StopWatch müssen Sie dazu nicht komplett übertragen. Es genügt, die Uhr des Rechners zweimal abzulesen und die Differenz zu berechnen. Schauen Sie nach, was die Bibliotheken dazu bieten!
- (3) Wettbewerb: Wessen Lösung ist am schnellsten? Senden Sie die reproduzierbaren Messergebnisse per E-Mail an karlheinz.hug@fh-reutlingen.de!

Aufgabe Ü1.5

Ausdrücke berechnen
mit rekursivem Abstieg

Mit dieser Aufgabe sollen Sie die grundlegende Methode für einen **Top-Down-Parser** üben und **Dialog-Ein-/Ausgabe** kennen lernen. Dazu wählen wir *I2P SS04 Ü4 A7: Ausdrücke berechnen mit rekursivem Abstieg*.

Studieren Sie die Musterlösung und übertragen Sie sie in die vier Sprachen! Falls die Entwicklungsumgebung kein GUI-Werkzeug umfasst, simulieren Sie die Lösung geeignet!

Aufgabe Ü1.6

Mengenklasse
verallgemeinern

Sinn dieser Aufgabe ist, **objektorientierte Sprachkonstrukte** kennen zu lernen und **dynamische Datenstrukturen** im **objektorientierten Stil** zu programmieren. Dazu wählen wir *I2P SS04 Ü5 A1: Mengenklasse verallgemeinern*.

- (1) Studieren Sie die Musterlösung und übertragen Sie sie in *Eiffel, Java, C#*!
- (2) In C++ müssen Sie besondere Sorgfalt für die **manuelle Speicherplatzverwaltung** der Halde aufwenden.
- (3) **Generizität**: In *Eiffel, C++ (template)* und ggf. *Java* entwickeln Sie eine Lösungsvariante mit einer generischen Klasse.

Aufgabe Ü1.7

Projekt: Ampelsystem/
Verkehrssimulation

Diese umfangreiche Aufgabe bearbeiten Sie als Projekt über das ganze Semester hinweg. Zweck ist, Sie auf das Software-Projekt im 5. und 6. Semester vorzubereiten. Dazu wählen wir *I2P SS04 PÜ1: Projekt: Ampelsystem/Verkehrssimulation*.

- (1) Studieren Sie die Musterlösung des **Ampelsystems** und übertragen Sie sie in die vier Sprachen! Gestalten Sie die Ausgabe geeignet nach den Möglichkeiten der Entwicklungsumgebung GUI- oder grafikbasiert!
- (2) Erweitern Sie die Lösung um eine (später sogar nebenläufige) **Verkehrssimulation** mit Fahrzeugen und Warteschlangen vor den Ampeln!

Ü2

Übungsblatt: Fragen zu den Sorter-Programmen

Bemerkung

Die Programme *Eiffel*: SORTER.e, SORTER_TEST1.e, SORTER_TEST2.e, C++: *Sorter.cpp*, Java: *Sorter.java* enthalten Aspekte, zu denen etwas zu lernen ist. Mailen Sie Ihre Antworten an karlheinz.hug@fh-reutlingen.de, damit sie zu Ihren Leistungen in *IP* zählen. Der Aufgabentyp eignet sich für Prüfungsaufgaben: Gefragt sind nicht Detailkenntnisse einzelner Sprachen, sondern die Fähigkeit, Sprachentwürfe anhand von Qualitätsmerkmalen zu beurteilen.

Aufgabe Ü2.1

Reihung als Standard-
bibliotheksklasse

Stroustrup empfiehlt, statt der unsicheren C-Reihungen die C++-Standardbibliotheksklasse *vector* zu verwenden. Passen Sie das C++-Programm *Sorter.cpp* entsprechend an!

Aufgabe Ü2.2

Modularisierung

Spalten Sie das C++-Programm *Sorter.cpp* auf in eine wiederverwendbare generische Schnittstelle (*template*) und einen Testteil!

Ü3

Übungsblatt: Objektorientierter Entwurf

Bemerkung

Gute Implementationen basieren auf guten Entwürfen; diese sind wichtiger als die gewählte Implementationssprache. Entwürfe sind sprachunabhängig, lassen sich aber in Sprachen formulieren. Neben der Beschäftigung mit Sprachkonzepten vergessen wir nicht, dass Software-Ingenieure die Fähigkeit auszeichnet, gute Entwürfe zu liefern.

Aufgabe Ü3.1

Globale Daten,
Module, Klassen

Eine fiktive hybride Programmiersprache mit Konstrukten für globale Daten, Module, Klassen bietet die Szenarien (a) - (g) der Programmierung. Dabei sind hier **globale Daten** außerhalb von Modulen und Klassen vereinbarte, überall sichtbare Daten. **Module** sind hier abstrakte Datenstrukturen, **Klassen** abstrakte Datentypen, d.h. Module und Klassen haben rein operationale Schnittstellen, sie exportieren die in ihnen vereinbarten konkreten Daten nicht, sondern verbergen sie und erlauben nur intern direkte Zugriffe darauf.

- (a) Zwei Module greifen auf dieselben globalen Daten zu.
- (b) Eine Klasse greift auf globale Daten zu, sodass jedes Objekt dieser Klasse auf diese Daten zugreifen kann.
- (c) Zwei Klassen greifen auf dieselben globalen Daten zu, sodass Objekte verschiedener Klassen auf diese Daten zugreifen können.
- (d) Ein Modul benutzt ein anderes Modul.
- (e) Ein Modul benutzt Klassen, indem es davon Objekte vereinbart, erzeugt oder benutzt.
- (f) Eine Klasse benutzt ein Modul, sodass ihre Objekte dieses Modul benutzen können.
- (g) Eine Klasse A benutzt eine Klasse B, sodass ein Objekt von A ein Objekt von B benutzen kann.

Untersuchen Sie für jedes der Szenarien (a) - (g) folgende Aspekte:

- (1) Zeichnen Sie ein Diagramm zur Veranschaulichung!
- (2) Wie kann man die Schnittstellen der Programmeinheiten spezifizieren und dokumentieren?
- (3) Wie ist die Konsistenz einer Programmeinheit zu gewährleisten? Bei (a) - (c): Welche Programmeinheit ist für die Konsistenz der globalen Daten zuständig?
- (4) Welche Programmeinheiten sind potenziell wiederverwendbar, welche nicht?
- (5) Beurteilen Sie das Szenarium hinsichtlich allgemeiner Prinzipien wie disziplinierte Strukturierung, Modularisierung und Lokalisierung und hinsichtlich struktureller Software-Qualitätsmerkmale!

Erläutern Sie, warum globale Daten seit Anfang der 1970er Jahre als softwaretechnisch problematisches Konzept gelten!

Modulare Sprachen der 1970er Jahre eliminierten globale Daten. Entsprechend kennen rein objektorientierte Sprachen seit den 1980er Jahren keine globalen Daten. Erläutern Sie, warum dies weder die Ausdrucksfähigkeit noch die Effizienz der Programmierung einschränkt!

Falls Sie die Möglichkeit globaler Daten für nützlich halten, erläutern Sie warum und nennen Sie ein Beispiel, das nicht mit anderen Programmierkonzepten besser lösbar ist!

Falls ein Software-Entwicklungsteam die Möglichkeit globaler Daten als problematisch einschätzt und die Sprache diese Möglichkeit bietet, so muss eine Programmierrichtlinie den Einsatz der Szenarien (a) - (c) verbieten. Wie ist das Einhalten dieser Richtlinie zu prüfen und zu garantieren? Welche Prüfungen kann welches Werkzeug durchführen?

Aufgabe Ü3.2

Objektorientierte
Programmiersprachen
und Datenabstraktion

Verlagert nach *Uebungen/Uebungsblatt I* Aufgabe Ü1.8 Datenabstarktion.

Aufgabe Ü3.3

Klassenbeziehungen

Die Abstraktionsarten **Klassifizieren durch Generalisieren und Spezialisieren** und **Zerlegen und Zusammensetzen** sind klar zu unterscheiden. Geben Sie jeweils die Beziehung zwischen den Dingen, Objekten oder Klassen in *UML*-Notation an! (Ankreuzaufgaben eignen sich nicht für Prüfungen.)

Tabelle Ü3.1 Klassenbeziehungen

Dinge, Objekte, Klassen	Aggregation, Komposition <i>A besteht aus B, A „hat ein“ B B ist Bestandteil von A</i>	Klassifizierung, Vererbung <i>A spezialisiert B, A „ist ein“ B B generalisiert A</i>
Wald, Baum		
Tanne, Baum		
Baum, Blatt		
Blatt, Titelblatt		
Hemd, Kleidung		
Hose, Kleidungsstück		
Prozessor, Rechner		
CD-ROM, Speicher		
Vogel, Spatz		
Spätzle, Menü		
Anweisung, Zuweisung ^a		
Zuweisung, Ausdruck ^a		
Musikinstrument, Trommel		
Trommel, Schlagzeug		
Atom, Molekül		
Pfifferling, Pilz		
Pils, Bier		

^a Die Antwort hängt von der gewählten Programmiersprache ab.

Aufgabe Ü3.4

Strukturmodell:
Klassifizierung oder
Zerlegung

Ordnen Sie die Begriffe

Dachbalken, Drucker, Bad, E/A-Gerät, Dachziegel, Bildschirm, Eingabegerät, Haus, Tintenstrahldrucker, Dach, Ausgabegerät, Badewanne, Laserdrucker, Küche, Scanner, Wohngeschoss

in eine Klassifizierungsstruktur und eine Bestandteilstruktur! Jeder Begriff taucht in genau einer Struktur auf.

Aufgabe Ü3.5

Strukturmodell:
Klassifizierung und
Zerlegung

Ordnen Sie die Begriffe aus dem Bereich „Verkehrsmittel“

Auslassventil, Automobil, Benzinmotor, Bremse, Dieselmotor, Einlassventil, Elektromotor, Fahrzeug, Getriebe, Lastkraftwagen, Lokomotive, Motor, Motorblock, Personenkraftwagen, Rad, Schienenfahrzeug, Straßenbahn, Tank, Ventil, Verbrennungsmotor, Zündkerze

in eine Klassifizierungsstruktur und eine Bestandteilstruktur! Ein Begriff kann in mehreren Beziehungen erscheinen.

Aufgabe Ü3.6

Strukturmodell:
Klassifizierung,
Zerlegung und
Assoziation

Ordnen Sie die Begriffe aus dem Bereich „Hochschule“

Assistent, AStA, Bachelor-Abschluss, Bachelor-Student, Beamer, Bibliothek, Cafeteria, Dekan, Dozent, Fachbereich, Fachbereichsrat, Fachschaft, Forschung, Hausmeister, Hochschule, Hochschulrat, Hörsaal, Hörsaalausstattung, Immatrikulation, Labor, Laboringenieur, Lehrbeauftragter, Lehre, Lehrveranstaltung, Master-Abschluss, Master-Student, Meister, Mensa, Personal, Praktikum, praktisches Studiensemester, Professor, Prüfung, Prüfungsamt, Prüfungsanmeldung, Prüfungstermin, Rechenzentrum, Rechner, Rektor, Rückmeldung, Sekretariat, Semester, Semesterferien, Seminar, Seminarraum, Senat, Student, Studentenschaft, Studienabschluss, Studiengang, Studiengangleiter, Studiengebühr, Studienplan, Studien- und Prüfungsordnung, Studium, Tutor, Urlaubssemester, Verwaltungsangestellter, Verwaltungsdirektor, Vorlesung, Vorlesungszeit, Zulassungsamt

in ein Strukturmodell mit Klassifizierungen, Bestandteilen und allgemeinen Beziehungen, die Sie treffend benennen!

Aufgabe Ü3.7

Von der Analyse zur
Implementation

Die objektorientierte Analyse kennt zwischen Klassen die Beziehungsarten Klassifikation, Komposition, Aggregation, Assoziation. Objektorientierte Programmiersprachen kennen die Beziehungsarten einfache Vererbung und Benutzung, manche auch mehrfache Vererbung und Schachtelung von Klassen.

Welche Analysebeziehungen werden auf welche Implementationsbeziehungen abgebildet? Diskutieren Sie Vor- und Nachteile unterschiedlicher Beziehungsarten in der Analyse und der Implementation!

Aufgabe Ü3.8

Abstrakte Klassen

Übersetzen Sie folgende Abschnitte aus [EIS90] S. 214!

„The abstract class mechanism supports the notion of a general concept, such as a *shape*, of which only more concrete variants, such as *circle* and *square*, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as objects representing a base class of a class derived from it. A class is abstract if it has at least one *pure virtual function*. A virtual function is specified *pure* by using a *pure-specifier* in the function declaration in the class declaration. A pure virtual function need be defined only if explicitly called with the *qualified-name* syntax.“

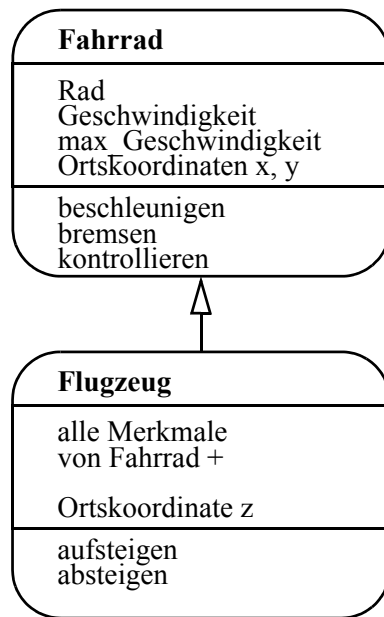
Aufgabe Ü3.9

Vererbung

Eine Zeitschrift einer bekannten Software-Firma, die objektorientierte Erweiterungen prozeduraler Programmiersprachen anbot, präsentierte einst das in Bild Ü3.1 dargestellte Beispiel für Vererbung.

Bild Ü3.1

Korrekt klassifiziert?



Erläutern Sie, warum dies dem Sinn und Zweck des Vererbungskonzepts widerspricht und geben Sie eine sinnvollere Struktur an (drei Klassen genügen dazu)!

Aufgabe Ü3.10

Leitlinien für Vererbung

Die folgenden Analyse-, Entwurfs- und Implementationsleitlinien postulieren, wie man Vererbung einsetzen soll. Begründen Sie zu jeder Richtlinie, warum sie sinnvoll oder praktisch nützlich oder warum sie problematisch oder inpraktikabel ist!

- (1) Wiederverwenden: Bilde nie eine neue Klasse, wenn Du eine existierende Klasse wiederverwenden kannst, sei es direkt, durch Benutzen oder Erben.
- (2) Klassifizieren: Setze Vererbung zum Generalisieren und Spezialisieren ein, d.h. um die „ist-ein“-Relation zu realisieren.
- (3) Enterben: Vermeide Vererbung, die nur dem Wiederverwenden von Code dient; verwende dafür Benutzung.
- (4) Herausfaktorisieren: Verbessere Vererbungsstrukturen, indem Du gemeinsame Merkmale von Unterklassen (Nachfolger-, abgeleitete Klassen) in gemeinsame Oberklassen (Vorgänger-, Basisklassen) verlagerst, ohne dabei die „ist-ein“-Relation zu verletzen.
- (5) Abstrahieren: Lass abstrakte Klassen die gemeinsamen Abstraktionen, Rollen und Merkmale all ihrer Unterklassen erfassen.
- (6) Hellsehen: Lass eine Klasse die Entwicklung zukünftiger Unterklassen nicht einschränken, solange diese die durch die Oberklassen gegebene Abstraktion konkretisieren.
- (7) Konzeptklassen bilden: Nutze abstrakte Klassen, um die zukünftige Entwicklung von Unterklassen geeignet zu ermöglichen und zu kontrollieren.
- (8) Schnittstellenklassen bilden: Nutze Vererbung, um polymorphes Ersetzen von Objekten der Unterklassen anstelle von Objekten der Oberklassen zu ermöglichen.

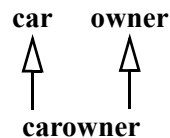
Aufgabe Ü3.11

Mehrfaches Erben

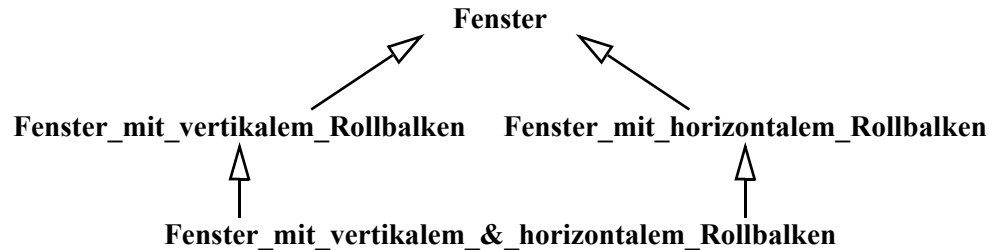
Entscheiden Sie für folgende Beispiele, ob mehrfaches Erben richtig eingesetzt ist oder nicht, begründen Sie Ihre Antwort und geben Sie ggf. eine bessere Lösung an!

Bild Ü3.2

Zweimal erben?

**Bild Ü3.3**

Diamantstruktur



Sollen bei den Klassen einer grafischen Oberfläche in Bild Ü3.3 die Merkmale Position, Breite, Höhe von **Fenster** in **Fenster_mit_vertikalem_&_horizontalem_Rollbalken** einmal oder zweimal vorkommen?

Aufgabe Ü3.12Entwurf von Klassen -
Gehaltsbuchhaltung

Eine Firma teilt ihre Mitarbeiter in Tarifliche, Außertarifliche und Abteilungsleiter. Das Gehaltsbüro kennt von jedem Mitarbeiter persönliche Daten, Adresse, Kontodaten. Das Gehalt eines tariflichen Mitarbeiters wird aus seinem Alter und seiner Einstufung in eine Tarifgruppe berechnet. Das Gehalt eines außertariflichen Mitarbeiters berechnet sich aus einem individuell vereinbarten Grundbetrag und einer von der Beurteilung des Vorgesetzten abhängigen Leistungszulage. Das Gehalt eines Abteilungsleiters hängt prozentual vom Umsatz der Abteilung ab.

- (1) Skizzieren Sie ein grafisches Strukturmodell für einen objektorientierten Entwurf der Gehaltsbuchhaltung! Erläutern Sie die Struktur!
- (2) Skizzieren Sie die Klassendefinitionen in einer geeigneten Spezifikationssprache! Konzentrieren Sie sich auf Syntax und Semantik der Schnittstelle! Vermeiden Sie, Implementationsentscheidungen vorwegzunehmen!
- (3) Erläutern Sie an diesem Beispiel den Begriff „abstrakte Klasse“ und wie Polymorphie und dynamisches Binden eingesetzt werden können!
- (4) Skizzieren Sie ein Codestück mit einem polymorphen Aufruf!

Ü4

Übungsblatt: Elemente von C++

Bemerkung

Diese Aufgaben eignen sich weder zu Prüfungsfragen, noch zeugt die Kenntnis ihrer Lösungen von softwaretechnischer Kompetenz. Wer jedoch tagtäglich mit C++ codiert, sollte einige Antworten wissen. Die *Vorlesung Informatik 3* bereitet nicht auf das Beantworten solcher Fragen vor, weil es wichtigere Themen gibt. Die Aufgaben sollen nur einen Eindruck davon vermitteln, welche Komplexität in Details stecken kann.

Ü4.1

Einfache Datentypen, Definitionen, Initialisierungen

Aufgabe Ü4.1

Nullen [Lip91b] S. 34

Erläutern Sie den Unterschied zwischen 0, '0', '\0', "0", 0.0 und NULL!

Aufgabe Ü4.2

Wahl des Zahlentyps [Lip91b] S. 29

Einer Variablen wird einer der drei möglichen Werte 0, 128, 255 zugewiesen. Diskutieren Sie Vor- und Nachteile der Deklaration der Variable als *double*, *int*, *unsigned char*, *char*!

Aufgabe Ü4.3

Definitionen [Lip91b] S. 37

Erläutern Sie die Bedeutung der folgenden Definitionen von Variablen! Identifizieren die illegalen darunter und versuchen Sie, sie zu korrigieren!

- (1) *int i;*
- (2) *const int ic;*
- (3) *const int *pic;*
- (4) *int *const cpi;*
- (5) *const int *const cpic;*

Aufgabe Ü4.4

Initialisierungen [Lip91b] S. 37

Welche der folgenden Initialisierungen sind legal, welche illegal? Warum?

- (1) *int i = 'a';*
- (2) *const int ic = i;*
- (3) *const int *pic = ⁣*
- (4) *int *const cpi = ⁣*
- (5) *const int *const cpic = ⁣*

Aufgabe Ü4.5

Besinnliches...

Gegeben ist

*typedef char *String;*

Ist

extern const String s;

äquivalent mit

*extern const char *s;*

oder mit

*extern char *const s;*

und warum?

Ü4.2

Zuweisungen

Aufgabe Ü4.6

Zuweisungen
[Lip91b] S. 37

Welche der folgenden Zuweisungen sind - die Definitionen von Aufgabe Ü4.4 vorausgesetzt - legal, welche illegal? Warum?

- (1) `i = ic;`
- (2) `pic = ⁣`
- (3) `cpi = pic;`
- (4) `pic = cpic;`
- (5) `cpic = ⁣`
- (6) `ic = *cpic;`

Aufgabe Ü4.7

Zuweisungen
[Lip91b] S. 29

Gegeben sind die Definitionen

`int ival = 1024; int *iptr; double *dptr;`

Welche der folgenden Zuweisungen sind legal, welche illegal? Warum?

- (1) `ival = *iptr;`
- (2) `*iptr = ival;`
- (3) `*iptr = &ival;`
- (4) `dptr = iptr;`
- (5) `ival = iptr;`
- (6) `iptr = ival;`
- (7) `iptr = &ival;`
- (8) `dptr = *iptr;`

Aufgabe Ü4.8

Zuweisungen
[Lip91b] S. 34

Gegeben sind die Definitionen

`int *ip1, ip2; char ch, *cp;`

Welche der folgenden Zuweisungen sind legal, welche illegal? Warum?

- (1) `ip1 = "All men are cremated equal!";`
- (2) `cp = 0;`
- (3) `ip1 = 0;`
- (4) `cp = &'a';`
- (5) `ip1 = ip2;`
- (6) `cp = '\0';`
- (7) `ip1 = '\0';`
- (8) `cp = &ch;`
- (9) `*ip1 = ip2;`

Aufgabe Ü4.9

Amüsantes und
Besinnliches...

Bertrand Meyer schreibt in [Mey97] S. 1134f über C++:

[...] this little excerpt from a pedagogical article by a recognized C and C++ authority, chair of the C standards committee of the American National Standards Institute and author of several respected C++ books as well as the *Dictionary of Standard C*, from whom I was at some point hoping to learn the difference between the C++ notions of reference and pointer (from Rex Jaeschke's C++ column in *DEC Professional*, November 1991):

While a reference is somewhat like a pointer, a pointer is an object that occupies memory and has an address. Non-const pointers can also be made to point to different objects at run time. On the other hand, a reference is an alias to an object and does not, itself, occupy any memory. Its address and value are the address and value of the object to which it is aliased. And while you can have a reference

*to a pointer, you cannot have a pointer to a reference or an array of references, nor can you have an object of some reference type. References to the **void** type are also prohibited.*

*References and pointers are not interchangeable. A reference to an **int** cannot, for example, be assigned to a pointer to an **int** or vice versa. However, a reference to a pointer to an **int** can be assigned a pointer to an **int**.*

I swear I tried to understand. I was almost convinced I got the hang of it, although perhaps not being quite ready for the midterm exam yet. ("Give convincing examples of cases in which it is appropriate to use: (1) A pointer only. (2) A reference only. (3) Either. (4) Neither. No notes or Web browsers allowed".) Then I noticed I had missed the start of the next paragraph:

From what we have seen so far, it may not be obvious as to why references indeed exist.

Übersetzen Sie die Abschnitte und kommentieren Sie sie!

Ü4.3

C-Reihungen

Aufgabe Ü4.10

Definitionen
[Lip91b] S. 43

Gegeben sind die Definitionen

```
int buf_size = 1024;
int get_size () {return 512;}
```

Welche der folgenden Definitionen sind legal, welche illegal? Warum?

- (1) `int ia [buf_size];`
- (2) `int ia [get_size()];`
- (3) `int ia [4 * 7 - 14];`
- (4) `int ia [2 * 7 - 14];`

Aufgabe Ü4.11

Initialisierung
[Lip91b] S. 44

Warum ist die folgende Initialisierung fehlerhaft?

```
char str [11] = "fundamental";
```

Aufgabe Ü4.12

Indexfehler
[Lip91b] S. 44

Das folgende Codestück enthält zwei Indexfehler. Identifizieren Sie sie!

```
int main () {
    const int array_size = 10;
    int ia [array_size];
    for (int ix = 1; ix <= array_size; ++ix) {
        ia[ix] = ix;
    }
}
```

Aufgabe Ü4.13

Fehlerquelle
[Lip91b] S. 48

Obwohl das folgende Programm ohne Fehlermeldung und Warnung übersetzt wird, enthält es einen schweren Fehler. Was ist das Problem? Wie können Sie es lösen?

```
char buf [] = "desaster";

int main () {
    char * ptr = 0;
    for (int i = 0; buf[i] != '\0'; ++i)
        ptr[i] = buf[i];
}
```

Bemerkung

Möchten Sie, dass die Software für Ihren Herzschrittmacher in einer Programmiersprache geschrieben ist, bei der weder der Übersetzer noch das Laufzeitsystem diese Arten von Fehlern erkennen können? Es sei daran erinnert, dass Stroustrup empfiehlt, statt der unsicheren C-Reihungen die generische Bibliotheksklasse *vector* der *Standard Template Library* (STL) zu benutzen.

Ü4.4

Aufgabe Ü4.14

Zuweisungen
[Lip91b] S. 72

Aufgabe Ü4.15

Ausdrücke
[Lip91b] S. 85

Aufgabe Ü4.16

Besinnliches...

Ausdrücke

Folgende Programme sind falsch. Warum? Wie korrigieren Sie sie?

- (1) `int main () {int i = j = k = 0;}`
- (2) `int main () {int i, j; int * ip; i = j = ip = 0;}`

In welcher Reihenfolge werden die Teilausdrücke in folgenden Ausdrücken ausgewertet? Welche Variablen ändern dabei ihren Wert? Wie sind die Teilausdrücke zu klammern, um der vermutlichen Intention des Programmierers zu entsprechen?

- (1) `!ptr == ptr->next`
- (2) `*process_table[current_nbr].state = blocked`
- (3) `ch = buff[bp++] != '\0'`
- (4) `~uc ^ 0377 & ui << 4`

Betrachten Sie folgende Programmfragmente.

```
class A {
public:
    char *f();
    ...
}

void g(char *s1, char *s2) {
    cout << s1 << endl << s2 << endl;
}

void h1() {
    A a;
    g(a.f(), a.f());
}

void h2() {
    A a;
    cout << a.f() << endl << a.f() << endl;
}
```

Der Autor eines Lehrbuchs wundert sich, dass die Funktionsaufrufe `h1()` und `h2()` unterschiedliche Wirkung (Ausgabe) hervorrufen; er räsoniert seitenlang über dieses „Dilemma“.

Unter welcher Bedingung können `h1`, `h2` unterschiedlich wirken, d.h. wo liegt hier das Problem? Unter welcher Bedingung nicht? Nennen Sie eine Programmierrichtlinie, die auch einen bestimmten Programmierstil kennzeichnet und die, wenn sie strikt eingehalten wird, Probleme obiger Art samt langatmiger Überlegungen verschwinden lässt!

Aufgabe Ü4.17

Amüsantes...

Mit welchen Definitionen sind folgende Zuweisungen korrekt und was bewirken sie dann?

- (1) `x=a/**/b;`
- (2) `x=a/ /**/b;`
- (3) `x/=a/**/*b;`
- (4) `x*=p**q;`
- (5) `x*=p*/**/*q;`

Ü4.5

Typregeln

Aufgabe Ü4.18

Typanpassung
[Lip91b] S. 91

Gegeben sind die Definitionen

char *c*; *short* *si*; *int* *i*, *j*, *k*; *long* *li*;
float *x*; *double* *d*;
unsigned char *uc*; *unsigned short* *usi*; *unsigned int* *ui*;

Welche der folgenden Zuweisungen sind unsicher wegen möglicher Wertebereichsverkleinerungen?

- (1) *si* = *i*;
- (2) *i* = *li*;
- (3) *si* = *uc*;
- (4) *li* = *ui*;
- (5) *ui* = *x*;
- (6) *i* = *usi*;

Identifizieren Sie den Datentyp folgender Ausdrücke!

- (7) 'a' - 3
- (8) *i* * *li* - *c*
- (9) *x* + *li* / *si*
- (10) *ui* + (*unsigned int*) *li*
- (11) *c* + *uc* + *li* + *ui*

Was passiert jeweils bei folgenden Ausdrücken?

- (12) *i* = *j* = *k* = 0;
- (13) *i* = *x* = *c* = 0;
- (14) *d* = *i* = *x* = 0;

Aufgabe Ü4.19

Typkonversion

Lippman schreibt in [Lip91b] S. 88:

The whole issue of type conversion can be quite bewildering, but it is something of which programmers must be aware.

Übersetzen Sie diesen Satz und erläutern Sie an drei kleinen Beispielen Probleme, auf die Lippman hinweisen will! Stellen Sie Programmierrichtlinien auf, die solche Probleme mindern sollen! Nennen Sie Entwurfsentscheidungen für eine Programmiersprache, in der solche Probleme nicht auftreten!

Ü4.6

Namensregeln

Aufgabe Ü4.20

Namensregeln

Stroustrup nennt - neben Kompatibilität mit C - folgende Ziele für Namensregeln in C++ [Str94a] S. 138ff:

- (a) We want to be able to do syntax analysis reading the source text once only.
- (b) Reordering the members of a class should not change the meaning of the class.
- (c) A member function body explicitly written inline should mean the same thing when written out of line.
- (d) Names from an outer scope should be usable from an inner scope (in the same way as they are in C).
- (e) The rules for name lookup should be independent of what a name refers to.

Drei von fünf Regeln des C++-ISO-Standards für den Gültigkeitsbereich von Namen, die in Klassen deklariert sind, lauten:¹

- (f) "The potential scope of a name declared in a class consists not only of the declarative region following the name's declarator, but also of all function bodies, default arguments, and constructor ctor-initializers in that class (including such things in nested classes)."
- (g) Reconsideration rule: "A name N used in a class S shall refer to the same declaration in its context and when re-evaluated in the completed scope of S. No diagnostic is required for a violation of this rule." (The completed scope of S consists of the class S, S's base classes, and all classes enclosing S.)
- (h) Reordering rule: "If reordering member declarations in a class yields an alternate valid program under (f) and (g), the program's behavior is ill-formed, no diagnostic is required."

Übersetzen Sie diese Abschnitte! Untersuchen Sie, ob folgende Programmfragmente korrekt, falsch, die „Nochmalbetrachtungsregel verletzend“ oder von „schlecht geformtem Verhalten“ sind! Im Falle von Korrektheit, worauf bezieht sich jeweils das x? Stellen Sie Programmierrichtlinien auf, die helfen sollen, angedeutete Probleme zu vermeiden!

- (1) `int x;`
`void f() {`
`int y = x;`
`int x = 1;`
`int z = x;`
`}`
- (2) `int x;`
`class B {`
`int f() {return x;}`
`int x;`
`}`
- (3) `typedef char *x;`
`class C {`
`x f() {x a = 0; return a;}`
`typedef int x;`
`}`
- (4) `const int x = 7;`
`class D {`
`int a [x];`
`int f() {return x;}`
`enum {x = 9};`
`}`
- (5) `class E {`
`x f();`
`void g() {x a; /* ... */}`
`typedef int x;`
`}`

¹ 1997 C++ Public Review Document (1997), 3.3.6 Class scope [basic.scope.class], <http://www.open-std.org/jtc1/sc22/open/n2356/>