Übungen, Stand: 08.01.2001

1.) Tippe die C++ Variante des "Hello World"-Programms ab und bringe es mit dem Compiler zum laufen.

2.) Schreibe ein Programm, das die Größen der fundamentalen Datentypen, ein paar Zeigertypen und ein paar Aufzählungen deiner Wahl ausgeben. Benutze dabei den sizeof-Operator (das ist eine Funktion, die in C/C++ fest integriert ist (formal ist sie als Operator definiert); sie gibt die Größe eines Objektes zurück).
3.) Schreibe die Deklaration für einen Zeiger auf einen char, ein Feld von 10 int, einen Zeiger auf ein Feld von Zeichenketten, einen Zeiger auf einen Zeiger auf ein char, einen konstanten int, einen Zeiger auf einen konstanten int und einen konstanten Zeiger auf einen int. Initialisiere jeden.
4.) Füge alle sinnerhaltenden Klammern in die folgenden Ausdrücke ein (Tipp: schreibe daneben, was jeweils bei gesetzten Klammern passiert):
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
5.) Schreibe eine Funktion, die die Fakultät berechnet und rekursiv arbeitet. Gib mit Hilfe dieser Funktion die Fakultäten der Zahlen 1 bis 10 aus.
6.) Schreibe eine Funktion, die die Werte zweier ints austauscht. Benutze int* (also Zeiger auf int) als Argumenttyp. Schreibe eine weitere Funktion, die int& (Referenz) als Argumenttyp verwendet.
7.) Was ist die Größe des Feldes str im folgenden Beispiel:
char str[] = "a short string";
Was ist die Länge der Zeichenkette "a short string"? (Tipp: Größe ist nicht gleich Länge)
8.) Definiere eine Tabelle mit den Monatsnamen des Jahres und der Anzahl der Tage in jedem Monat. Gebe diese Tabelle aus. Tue dies genau zweimal: einmal mit einem Feld von char für die Namen und einem Feld für die Anzahl der Tage und einmal mit einem Feld von Strukturen, wobei jede Struktur den Namen des Monats und die Anzahl seiner Tage enthält.
9.) Definiere ein Feld von Zeichenketten, wobei die Zeichenketten die Monatsnamen enthalten. Gebe diese Zeichenketten aus. Übergebe das Feld an eine Funktion, die die Zeichenketten ausgibt.
10.) Lese eine Folge von Wörtern aus der Eingabe (analog zu cout ist die Eingabe über cin>>variablenname möglich). Benutzer Quit als das Wort, das die Eingabe beendet. Gebe die Wörter in der Reihenfolge aus, in der sie eingegeben wurden. Gib kein Wort mehrmals aus. Modifiziere das Programm so, dass es die Wörter sortiert (Tipp: verwende die Klasse string - das ist eine gute Gelegenheit etwas mit dieser Klasse vertrauter zu werden :-)).
11.) Schreibe eine Funktion cat(), die zwei C-Strings (also Arrays von chars) als Argumente bekommt und einen C-String zurückgibt, der die Konkatenation der beiden Argumente ist. Benutze new um Speicher für das Ergebnis zu bekommen (Tipp: überlege dir genau, wie du die Strings übergibst, bzw. wie du sie zurücklieferst). Führe dieselbe Aufgabe mit der Klasse string durch.
12.) Etwas größeres Projekt: Schreibe einen kleinen Taschenrechner. Das Programm wird mit dem zu berechnenden Ausdruck aufgerufen, wobei es sich um einen "einfachen" Ausdruck handeln soll (z.B. 1+3; Eingabe an der Kommandozeile lautet dann z.B. rechner 1+3), von dem das Programm dann das Ergebnis berechnet. Der Taschenrechner soll das Ergebnis für +, -, und * berechnen. Verwende in deinem Programm die Parameter von main() und zur Syntaxanalyse switch-case.
13.) Betrachte:

```
struct Tnode{
        string wort;
        int anzahl;
        Tnode* links;
        Tnode* rechts;
}
```

a) Schreibe eine Funktion, um neue Wörter in einen Baum von Tnodes einzufügen, Schreibe eine Funktion, die einen Baum von Tnodes ausgibt. Schreibe eine Funktion, die einen Baum von Tnodes mit alphabetisch sortierten Wörtern ausgibt.
b) Arbeite so mit Dateien, dass ein Programm, das ein Tnode verwenden möchte, nur noch eine Header-Datei einzubinden braucht und eine entsprechende Objektdatei hinzu linken muss. Schreibe zur Überprüfung ein (kurzes) Beispielprogramm, das dies leistet.
c) Ändere Tnode so ab, dass nur (ein) Zeiger auf ein beliebig langes Wort, das als Feld von char im Speicher

mit new angelegt wird, verwendet wird.
14.) Konstruiere einen (kleinen) Fall, indem eine Header-Datei möglicherweise mehrmals eingefügt würde. Schreibe diesen Fall auf und behebe ihn mit den entsprechenden #if defined(...) Anweisungen. Schreibe deine Lösung als Programm auf und versuche es zu compilieren (einmal mit Zyklus, einmal mit #if defined...).

Klassen:
15.) Schreibe selbst eine Klasse Datum, die die Datumseingabe von der Eingabe (cin) liest und auch wieder ausgibt. Ergänze weiterhin eine Elementfunktion, mit der man einen Tag, einen Monat und ein Jahr auf das Datum aufaddieren kann.
16.) Reimplementiere die Klasse Datum mit einer "Tage seit dem 01.01.1970"-Repräsentation
17.) Definiere einige Klassen, die Zufallszahlen mit einer bestimmten Verteilung (z.B. gleichmäßig oder exponentiell), erzeugen. Jede Klasse soll die Parameter für die Verteilung über den Konstruktor erhalten. Eine Funktion ziehe() soll den nächsten Wert liefern.
18.) Schreibe die Struktur Tnode (siehe oben) neu als Klasse mit Konstruktoren, Destruktor etc. Definiere einen Baum aus Tnodes als Klasse mit Konstruktoren, Destruktoren etc.

Überladene Operatoren:
19.) Erweitere die Klasse Datum (siehe oben) um Operatoren + und -, die jeweils ein anderes Datum addieren bzw. subtrahieren. Überprüfe die Korrektheit der Klasse mit entsprechenden Beispielwerten.
20.) Definiere eine Klasse INT, die sich genau wie ein int verhält. Tipp: definiere einen Operator INT::operator int().
21.) Definiere eine Klasse vector, die einen Operator [] enthält, so dass man auf ein vector-Element wie auf ein Array zugreifen kann. Verwende als Elemente, die vector enthalten kann int.

Abgeleitete Klassen:
22.) Für die Klasse

```
class Basis{
public:
        virtual void ichBin() {cout << "Basis\n";}
}
```

leite zwei Klassen ab und definiere für jede Klasse ichBin() so, dass der Name der Klasse ausgegeben wird. Erzeuge Objekte dieser Klasse und rufe ichBin() für sie auf. Weise Zeiger auf Objekte der abgeleiteten Klassen an Basis* -Zeiger zu und rufe ichBin() über diese Zeiger auf.
23.) Für die Klassen Kreis, Quadrat und Dreieck, jeweils abgeleitet von der abstrakten Klasse Form, soll eine Funktion ueberlappt() definiert werden, die für zwei Form* Argumente passende Funktionen aufruft, um festzustellen, ob sich die beiden überlappen. Es ist dazu notwendig, passende (virtuelle) Funktionen zu den Klassen hinzuzufügen. Halte dich nicht damit auf, den Code zu schreiben, der auf Überlappung prüft; achte nur darauf, dass sie die richtigen Funktionen aufrufen.

Ausnahmebehandlung:
24.) Schreibe ein Programm, das eine Ausnahme in einer Funktion wirft und sie in einer anderen auffängt.
25.) Schreibe ein Programm, das aus Funktionen besteht, die einander bis zu einer Aufruftiefe von 10 aufrufen. Gebe jeder Funktion ein Argument, das bestimmt, auf welcher Ebene eine Ausnahme geworfen werden soll. Lasse mit main() diese Ausnahmen fangen und ausgeben, welche Ausnahme gefangen wurde. Vergesse nicht den Fall, in dem eine Ausnahme in der Funktion gefangen wird, die sie geworfen hat.
26.) Schreibe eine Funktion, die einen Wert basierend auf ihrem Argument entweder per return liefert oder diesen Wert als Ausnahme wirft.
27.) Ausnahmen sind Objekte einer Klasse. Diese kann auch aus einer ganzen Fehlerklassen- Hierarchie stammen, d.h. von einer anderen Klasse abgeleitet sein. Denke dir eine Beispiel-Fehlerklassenhierarchie (z.B. Mathematische Fehler wie Overflow, Underflow, Division durch Null, ...) aus. Schreibe ein Beispielprogamm, das einen Fehler aus der untersten Hierarchieebene wirft und prüfe, was passiert, wenn du anstelle dieses Fehlers die Basisklasse der Hierarchie fängst.

Templates:
28.) Schreibe ein Template, das einen Stack repräsentiert, der beliebige Elemente verwalten kann. Zur Überprüfung schreibe eine Klasse Element, mit Hilfe dessen Objekten du das Template testest.
29.) Schreibe ein vector-Template (siehe auch weiter oben bei den Operatoren), das beliebige Elemente verwalten kann und Range- und Size- Ausnahmen besitzt (meint: wenn man über [] auf nicht vorhandene

Elemente zugreifen will etc.).
30.) Schreibe ein leseZeile()-Template für verschiedene Zeilenarten, z.B. Artikel-, Anzahl-, Preis-Zeilen.

STL:
31.) Definiere eine Klasse telnummer, die Telefonnummern speichern kann. Instanziiere die Klasse list (dazu die Header-Datei <list.h> einbinden), die Daten vom Typ telnummer aufnehmen kann. Füge als Beispiel mehrere Telefonnummern in die Liste ein. Durchlaufe sie dann einmal mit Hilfe von Iteratoren in einer While-Schleife, und danach ein zweites Mal mit Hilfe der Funktion for_each (dazu die Header-Datei <algorithm> einbinden).
Leite nun von der Klasse list<telnummer> eine Klasse tel_liste ab, die eine zusätzliche Funktion für die Ausgabe der Telefonnummern besitzt.
32.) Definiere eine Queue, die nur zwei Objekte der Klasse stack verwendet.

Anmerkung: Weitere Übungen zur Standardbibliothek (insbesondere was die Dateioperationen/Streams betrifft) wurden nicht mit aufgenommen, sondern durch Übungen zu LEDA ersetzt. Die STL funktioniert auf dieselbe Weise wie LEDA, nur mit entsprechend anderen Klassen. Man sollte sich die Klassen und die Hierarchie einmal in einem entsprechenden Buch angeschaut haben.

LEDA:
33.) Implementiere den Typ stack mit Hilfe des Ledatyps array (für allgemeine Typen als Elemente).
34.) Implementiere den Typen queue mit Hilfe des LedaTyps array (für allgemeine Typen als Elemente).
35.) Implementiere den Typen stack mit Hilfe (Vererbung) des LedaTyps sList.
36.) Implementiere den Typen queue mit Hilfe (Vererbung) des LedaTyps sList.
37.) Implementiere eine Funktion reverse, die als Parameter eine list bekommt und als Rückgabe eine list erzeugt, die die Elemente der Parameter-list umdreht.
38.) Implementiere eine Funktion conc, die zwei Parameter vom Typ list bekommt und als Rückgabe eine list. erzeugt, die die Elemente der beiden list konkateniert.
39.) Implementiere eine Funktion merge, die zwei Parameter vom Typ list und eine Compare-Funktion bekommt (die eine lineare Ordnung auf den Listenelementen erzeugt) und die Listenelemente geordnet in eine neue Liste einfügt.
40.) Implementiere eine Funktion quicksort bezüglich des Typs array.
41.) Implementiere eine Copy-Funktion, die eine Kopie eines Graphen (als Typ gegeben) erzeugt und dabei alle Kanten umdreht.
42.) Gegeben sei graph als Typ. Implementiere eine Funktion strongComponents, die einen graph erhält und starke Zusammenhangskomponenten berechnet. Die Rückgabe soll ein node_array sein, wobei die Nummer der Zusammenhangskomponente vermerkt ist.
43.) Gegeben sei graph als Typ. Implementiere eine Funktion, die überprüft, ob der übergebene graph symmetrisch ist und falls dem so ist, ein edge_array erzeugt, das für jede Kante sein Pendant enthält.
44.) Erstelle einen parametrisierten Graphen. An den Knoten soll ID, an den Kanten soll der eine Fahrzeit und eine Liniennummer vermerkt werden. Erzeuge einen Multigraphen (mit ein paar paar beliebigen Werten). Implementiere eine Funktion, die zwei Argumente (HaltestellenIDs) bekommt, und nachschaut, ob diese adjazent sind und ggf. die einzelnen Linien und deren Fahrzeiten (sortiert nach Fahrzeit) ausgibt.