„Divide et impera.“

Rekursivität:

Pseudocode mit Modul A(D): **IF** (A(D) „triviales Problem“)
THEN
Gebe triviale Lösung explizit an.
ELSE
a) teile D in [möglichst disjunkte] D_1,D_2, ..., D_k auf und berechne $A(D_1)$, $A(D_2)$, ..., $A(D_k)$.
b) setze Teillösungen zur Gesamtlösung $A(D)$ zusammen.

LIFO - Last In First Out
FIFO - First In First Out

Sortierung:

Tree Sort Baum erstellen, wobei jeder neue Buchstabe von der Wurzel an beginnend durchgeschoben wird (links lang, wenn $\leq$, rechts lang, wenn $>$), bis er ein Blatt darstellt. Ausgabe erfolgt dann von links unten an aufsteigend für alle Äste.

Merge Sort Liste in 2 Hälften teilen und jede sortieren ($\rightarrow$ rekursiv); danach beide zusammenfügen.

Quick Sort Man bringt ein Element an seine bestimmungsgemäße Stelle und ruft den Algorithmus rekursiv auf, so dass er das gleiche für die Teile rechts und links von dem schon gefundenen Element macht.

Heap Sort Vollständigen Binärbaum aus Daten erstellen, dabei jedesmal darauf achten, dass die Größe der Elemente nach oben im Baum zunimmt. Dann das erste Element herausnehmen (das größte), das letzte an dessen Stelle setzen und aus diesen Elementen einen den Anfangs-Regeln entsprechenden Heap erzeugen, wieder dessen erstes Element herausnehmen, das dann letzte nach oben bringen, ...

Merge- , Quick- und Heap-Sort stellen die schnellsten Sortieralgorithmen dar (vgl. Informatik I).

Globale Variablen:
static int var = 0; legt eine Variable statt auf dem Stack (Stapel) auf dem Heap (Halde/Haufen) an; diese hat dann ewige Lebensdauer und wird nur einmal initialisiert.

Strukturen:
Definition: ; Umgang: .

```
struct name {
        int i,j;
        double x,y,z;
        char title[41];
};
```

```
name var1,var2;

var1 = {1,2,3.14,2.718,0,"..."};

var1.x = 25;

var2 = var1              // echte Kopie
```

Pointer:

```
name *ptr_var1 = &var1;

ptr_var1->z = 1;          // gleichbedeutend zu "(*ptr_var1).z = 1;"
```

Typendeklaration:
typedef char* string; teilt dem Compiler mit, dass alle Variablen des Typs string eigentlich **char*** sind.

Dateien:

```
FILE *ptr_file; char *name[81];
fgets name;
ptr_file=fopen(name,"w+");
if (!ptr_file) { printf("Unable to open file!"); return 1; }
fprintf(ptr_file,"Text und so.");
fclose(ptr_file);
```

Standardgeräte: stdin - Tastatur, stdout - Terminal, stderr - Fehlerausgabe (Terminal), stdprn - Drucker, stdaux - Serieller Anschluss.

HOM - Haupt-Ordnungs-Merkmal

Divisionsrestverfahren zur **Dateiorganisation**:

- Größe des Raumes der Datensätze muss bekannt sein. 25000 Studenten
- Großzügig ca. 20% zusätzlich. 30000 Speicherplätze
- Festlegen des Divisors (nächstgrößere Primzahl). 30011
- Anlegen einer Datei mit dem Divisor entsprechenden Datensätzen. 30011
- Füllen der Datei mit Sätzen und zugeordneten *HOM* $\xrightarrow{\text{Umrechnen}}$ Nummer im Adressraum zuweisen (Nummer modulo Divisor $\Leftrightarrow$ Satznummer).
- Kollisionsproblem: zwei *HOM* auf gleiche Satznummer abgebildet. $\rightarrow$ Suche nach nächstem freien Platz.

[Besser sind dann lineare Listen mit Folgeadresse in jedem Datensatz.]

Makros:

1. einfache Makros: **#define** NAME ersatztext — **#define** PI 3.14
2. parametrisierte Makros: **#define** NAME(parameter) ersatztext — **#define** QUADRAT(x) ((x)*(x))

Listing 1: Einfach verkettete Liste

```
typedef struct node *Ref;                  // Nur Pointer dürfen vor ihrer Deklaration schon
typedef struct node {                         benutzt werden.
        int key;
        Ref next;
} NODE;
Ref p,q,First;
First = (Ref) malloc(sizeof(NODE));
First->key = 1;                            // Inhalt
p=First;
for (int i = 2; i <= n; i++)
        q = (Ref) malloc(sizeof(NODE));
        q->key = i;                        //Inhalt
        p->next = q;
        p = q;
}
p->Next = NULL;                            // NULL-Pointer
```

C++:

Deklarationen dürfen überall stehen.

#include <iostream> , kein „.h“ mehr nötig.

<<, >> , Text anfügen, auslesen.

#include <fstream> , ifstream [input-file-stream] und ofstream [output-file-stream].

Kommentare: „/* ... */“ oder „ ...; // ...“.

Dynamische Datenobjekte:

```
int* p;
p = new int;
*p = 15;

delete p;
```

```
int* pa = new int[40];
int** pb = new int*[40];

delete []pa;
delete []pb;
```

Referenzen:

```
int fkt(double& a) {
        a = 3*4;
        return 0;
        }

double b;
fkt(b);
```

; bzw. **int** b; **int**& a=b;
Dabei erhält a den gleichen Pointer wie b [&a = &b;];
dies muss bei der Initialisierung bzw. dem Funktionsaufruf passieren.
Dies macht der Compiler.

Default-Werte:
in Prototypendeklaration angeben [**void** fkt(**int** a, **int** b, **char***="DEFAULT");] ; interpretiert der Compiler.

Abstrakte Datentypen: Datentypen + Funktionen **Klasse**: konkreter, abstrakter Datentyp.

```
class blub {
        private:
        // Funktionsprototypendeklarationen + Variablen
        public:
        // Funktionsprototypendeklarationen + Variablen
        };
blub Objekt1, Objekt2;
```

Hier nur Prototypendeklarationen, da diese in alle Objekte kopiert werden; diese wie folgt: **void** blub::fkt1() {...;} .
[:: - „Scope-“ / „Sichtbarkeitsoperator“]

Als Anwender hat man nur Zugriff auf die **public**-deklarierten Variablen und Funktionen; als Konstrukteur auf alle.

Den Objekten kann man „Aufträge erteilen“: Objekt1.fkt1(); .

In jeder Memberfunktion gibt es einen klassenspezifischen Zeiger: **this** , der auf das aktuelle [↔ wo die aufgerufene Memberfunktion drin prototypendeklariert ist] Objekt zeigt.

Konstruktor / Destruktor:

- Zum Initialisieren / sauberen Löschen der Objekt-Daten [meist **public**].
- Konstruktor: Name identisch mit Klassennamen; sofern keiner explizit programmiert, gibt es immer den Systemkonstruktor [reserviert nur Speicher].
- Abhängig von zugewiesenen Wert verschiedene Konstruktoren.
  [„Typumwandlungskonstruktor“, blub Objekt1(18,05,2013); oder blub Objekt1("18.05.2013");]

Konstanten in Objekten: **class** object {...; **const int** blub; ...;}; , mit Konstruktor
object (..., bla ,...): blub(bla) {...}; ; Reihenfolge beachten.

Kopierkonstruktor: object(**const** object& A); [→ kein Selbstaufruf!].

Impliziten Aufruf von Konstruktoren verhindern: **explicit** object (...); .

Destruktor: ~object(); ;
wird beim Verlassen des Gültigkeitsbereichs der Variablen vom System automatisch aufgerufen.

Implementierung: object::object (...){...}; , object::~object (){...}; .

Mit **friend class** ... , bzw. **friend** function ... kann man bestimmten Funtktionen den Zugriff auf **private**-Daten erlauben.

Liste von Listen [Matrix, Tensor, ...]: vector<vector<**int**> > A(m, vector<**int**> (n));

void sort (RandomAccessIterator first, RandomAccessIterator last); sortiert first bis last in aufsteigender Reihenfolge.

Wahrheitswerte: **bool**-Datentyp mit **true** [1] bzw. **false** [0].

Komplexe Zahlen: complex-Datentyp [z.B.: complex<**double**> a(1.0,2.0); [$a = 1 + \mathrm{i}\,2$]],
mit Funktionen: $+, -, *, /$,exp,log,conj,....

Namensbereiche: Gültigkeit von Variablen nur in den Namensbereichen, in denen sie deklariert wurden!
[**namespace** A ..., **using namespace** std;, std ist der Standardnamensbereich]

Überladen von Operatoren:

1. Memberfunktion: Returndatentyp klassenname::**operator** $\otimes$ (Argumentliste) {...;};
2. gewöhnliche Funktion: Returndatentyp **operator** $\otimes$ (Argumentliste) {...;};

Fast alle dem Compiler bekannten Operatoren überladbar [+, −, ∗, /, [], (), <<, >>, &, &&, |, ||, <, <=, >, >=, ==, +=, −=, ...; **nicht**: ?:, ::, ., .∗, **typeid**, **sizeof** und die C++-Cast-Operatoren.] .

Auf Verwendungszweck achten! [Ob Rückgabe oder nicht; Rückgabe als Referenz oder Kopie [→ Gültigkeitsbereich der Variablen]; ...]

Definiert man **const**-Objekte, so können auch nur entsprechende Memberfunktionen darauf zugreifen:
returntype fkt (...) **const** {...; *// nur lesen* }; .

Parametrisierung von Klassen mit **Templates**:

Sofern für verschiedene Datentypen [z.B. **int**,**double**,**unsigned int**,...] eine Klasse erstellt werden soll, die für alle das gleiche macht, dann kann man, statt für jeden Datentyp einzeln eine Klasse zu erstellen, dies über Templates [„Schablonen“] machen.

template <**class** T> initialisiert T als Template; dann kann man ganz normal Objekte deklarieren, mit Daten vom Typ T.

Die Deklaration der Memberfunktionen erfolgt dann wie folgt:
template <**class** T> returntype classname<T> :: memberfunction (parameter) {;}

Der Aufruf dann gemäß: classname<usable datatype>

Da der Compiler jedoch nichts mit Templates anfangen kann, muss die Klassendeklaration in jedes Programm kopiert und kann nicht vorkompiliert werden.

Funktionstemplates:

```
template <class T>
const T& max(const T& a, const T& b) {
  if (a > b) {
        return a;
  }
  else {
        return b;
  }
}
```

Vererbung:

Eine abgeleitete Klasse enthält zur Oberklasse zusätzliche Informationen [→ „ *abgeleitete* ist ein *Oberklasse*“]

Deklaration gemäß: **class** oberklasse : **public** unterklasse { ...;} ; beim Aufruf zuerst die Konstruktoren der Oberklasse benützt.

Statt **private** [nur eine Vererbungsebene] **protected** benützen.

Bei Vererbung virtuelle Funktionenen [**virtual void** fkt () {;}] und Objekt-Benutzung über Pointer; dann werden gleichnamige Memberfunktionen vererbter Klassen dem Objekt entsprechend aufgerufen.

Kopie eines vererbten Objektes auf ein Oberklasse-Objekt ist erlaubt, dabei gehen aber die zusätzlichen Informationen verloren; andersherum ist es nicht erlaubt.

Auch die Kopie eines Pointers eines vererbten Objektes auf den eines Oberklasse-Objektes geht, wobei keinerlei Information verloren geht.

heterogene Listen:
Abstrakten Basisdatentyp als Standardlistenelement und dann abgeleitete davon als heterogene Elemente. Virtuelle Funktionen zum gleichnamigen Zugriff auf alle Elemente.

C-Befehle

- Variablentypen: **int**, **double**, Image
- printf("control␣string", argument list);
- scanf("control␣string", &argument list);
- InfImgFile(name, dimx, dimy, ch);
- name = NewImg(dimx, dimy, 255);
- name = NewImg(pic, 0/1);
- ClearImg(name);
- Show(ON,name,"..."); / Show(OFF,name);
- GetVal(name, x, y); , PutVal(name, x, y, value);
- FreeImg(name);, ReadImg(filename, name);, WriteImg(name, filename)
- GetChar();
- Show(OVERLAY,pic,mark,"Schwerpunkt"); Marker(DEFAULT,sx,sy,farbe,groesse,bild);
- **#include** <math.h> ⇒ x = sqrt (...);
- **#include** <stdio.h> ⇒ d=InputD("\nText:␣");, i=Input("\nText:␣"); Printf("\nText:␣");
- GetChar()
- Casten: (type)(expression)
- **sizeof**()
- condition ? expression1 : expression2
- **int** sprintf = (c_string,"control␣string", argument list);
- **void**∗ malloc(m ∗ bytes);
- **struct** name {...;} var;
- **typedef** type name;
- ptr = fopen(filename,mode);, fclose(filename);, fprintf(filename,text,var);.
- ptr = (ptr_type) malloc(#Bytes); ...; free(ptr);

C++-Befehle

- Casten: type(expression)
- Scope-Operator ::
- Zuweisen: type/**class** variable(values);
- String-Member-Fkt.: clear(), size(), begin(), end(), iterator, c_str()
- Vector-Member-Fkt.: size(), push_back(value), pop_back(), clear()
- Complex-Member-Fkt.: real(), imag()
- **public**, **private**/**protected**
- **new**, **delete** ; **new**[], **delete**[]
- **friend class**, **friend** function ()
- **using namespace** ...;
- Returndatentyp **operator** ⊗ (Argumentliste) {...;};
- **template** <**class** T>
- **virtual void** fkt () {;}