Grundgebiete der Informatik II
im Elektrotechnik–Grundstudium

ALGORITHMEN UND

PROGRAMMIERTECHNIKEN

Prof. Dr.–Ing. M. Nagl

Dipl.–Inform. S. Becker

Dipl.–Inform. B. Böhlen

Lehrstuhl für Informatik III

Ahornstr. 55

RWTH Aachen

Inhalt

Anhang I: Syntax
- Schlüsselwörter
- Syntaxbeschreibung (kontextfreie) von C++
- Hinweise zur kontextsensitiven Syntax

Anhang II: Graphische Notationen für Programme
- Flußdiagramme
- Struktogramme

Anhang III: Arbeiten am CIP–Pool in C++
- Hinweise für Rechner– und Compilerbenutzung
- Hinweise für C++
- Beispiel und Übungen für C++

Anhang IV: Übungsaufgaben zum Selbststudium

Anhang V: Klausuraufgaben

Anhang VI: Glossar

1 Zielsetzung, Einordnung, Literatur

Zielsetzung

- Konstrukte heutiger Programmiersprachen für
  Ablaufkontrolle (Konstrollstrukturen)
  Datenstrukturierung (Datentypkonstruktoren)
  Programmgrobstrukturierung
- Syntaxnotation
- Systematisches Programmieren im Kleinen
  Problem ➧ Algorithmus ➧ Programm
  ➧ Betrachtung (Test, Effizienz)
- Standardkenntnisse: Suchen und Sortieren
- Modularisierung:
  Übergang zum Programmieren im Großen:
  Bausteine, Zusammensetzung, Wiederverwendung
- Datenabstraktion und Formulierung durch Module
- Standardkenntnisse Datenstrukturen:
  Listen, Bäume, Graphen: Wie realisiert man sie?
  Wie verwendet man sie?
- Grundzüge der Objektorientierung
- Nutzung der weitverbreiteten Programmiersprache C++
- Ausblick Softwaretechnik

dabei: Arbeiten mit Arbeitsplatzrechnern (CIP–Pool), Arbeiten mit mod. Werkzeugen

Veranstaltung legt Grundlage für

- Programmierpraktikum (anschließend)
- vergl. Darstellung höherer Programmiersprachen
- Effiziente Algorithmen: genauere Effizienzbetrachtung
- theor. Konzepte für Programmierung/Programmiersprachen
- andersartige Programmiersprachen
  (Lisp, Smalltalk, Prolog)
- Softwaretechnik
- maschinennahe Programmierung
- nebenläufige Programmierung
- Compilerbau
- Datenbanksysteme
- Betriebssysteme

und beliebige Anwendungsprogrammierung.
Wichtig in erster Linie für Vertiefung in Technischer Informatik

Warum C++?

C++

– Objektorientierung und andere Konstruktionsprinzipien
– weite Verbreitung für technische Anwendungen
– C hat Features für systemnahe Programmierung (z.B. Adreßrechnung)
– Auf vielen Plattformen verfügbar

C++ und Methodik

– auch in C++ läßt sich sauber programmieren bzw. ein Programm strukturieren
– Erklärung ist nicht C++, sondern methodische Verwendung (z.B. Syntax, z.B. Sprachgebrauch): Einschränkung, Disziplin
– Transliterationstabelle?

Literatur

1. Standardliteratur zur Sprache C++

Bjarne Stroustrup: Die C++ Programmiersprache, Addison–Wesley, 1998

Margaret A. Ellis / Bjarne Stroustrup: The annotated C++ reference manual, Addison–Wesley, 1995

Stanley B. Lippman: C++ primer, Addison–Wesley, 1991

Martin Schader / Stefan Kuhlins: Programmieren in C++, Springer, 1998

2. OO Sicht auf C++

H. M. Deitel / P. J. Deitel: C++ How to Program, Prentice–Hall Inc., 1997

Nicolai Josuttis: Objektorientiertes Programmieren in C++: von der Klasse zur Klassenbibliothek, Addison–Wesley, 1994

3. Beschreibung der Standardbibliothek

Nicolai Josuttis: Die C++–Standardbibliothek, Addison–Wesley–Longman, 1996

Martin Schader / Stefan Kuhlins: Die C++–Standardbibliothek, Springer, 1999

David R. Musser / Atul Saini: STL tutorial and reference guide: C++ programming with the standard template library, Addison–Wesley, 1996

4. Vergleich zu anderen Programmiersprachen

László Böszörményi / Carsten Weich: Programmieren mit Modula–3, Springer 1995

Greg Nelson: Systems programming with Modula–3, Prentice Hall, 1991

Manfred Nagl: Softwaretechnik und Ada 95, Vieweg, 1999

5. Architektursicht

Grady Booch: Object–oriented analysis and design with applications, Benjamin/Cummings, 1994

James Rumbaugh / Michael Blaha / William Premerlani / Frederick Eddy / William Lorensen: Objektorientiertes Modellieren und Entwerfen, Prentice–Hall, 1993

Erich Gamma / Richard Helm / Ralph Johnson / John Vlissides: Design Patterns: Elements of Object–Oriented Software, Addison–Wesley, 1994

Frank Buschmann / Regine Meunier / Hans Rohnert / Peter Sommerlad / Michael Stal: Pattern–orientierte Software–Architektur, Addison–Wesley–Longman, 19

Manfred Nagl: Softwaretechnik: Methodisches Programmieren im Großen, Springer, Berlin, 1990

6. Programmierstil, Empfehlungen

James O. Coplien: Advanced C++ programming styles and idioms, AT&T Bell Telephone Laboratories, 1992

Mats Henricson / Eric Nyquist : Industrial Strength C++, Prentice Hall, 1997

2 Hilfsmittel zum Programmieren im Kleinen

Erstes Beispiel

C++ – Programm

```
#include <iostream.h>
int main()
{  int summe, a, b;
   // Lies die Zahlen a und b ein
   cout << "a und b eingeben:";
   cin >> a >> b;

   /* Berechne die Summe beider
       Zahlen */
   summe = a+b;

   // Zeige das Ergebnis auf dem
   // Bildschirm an
   cout << "Summe=" << summe << endl;
   return 0;
}
```


Begriffe:

↓ für Info an das Hauptprogram

Hauptprogramm `main( ) { ... }`

Block { ... }

Kommentare `//` ..... bis Zeilenende

`/*` .....

..... `*/`

Einbindung Ein– / Ausgabefunktionen: Datei `iostream.h`

Variablendeklaration von `summe, a, b,` Bezeichner, Typ

Programme können Werte an andere geben, sonst `void`

Vordefinierter Datentyp `int`

Standardeingabe, –ausgabe `cin >> a, cout << summe`

Zeichenkettenliteral `"Summe="`

Zuweisungszeichen `=` , Ausdruck, Wertzuweisung,

Anweisungsfolge

Wie ist ein Programm aufgebaut? (Syntax)
Welche Berechnung führt es aus? (Semantik)
Wie wird es aufgetragen; Ist es lesbar;
Wie schnell ist der Übersetzer? (Pragmatik)

} folgender Abschnitt

Zusammenfassung: Semiotik

Welche Hilfsmittel zur Ausgestaltung von Programmen existieren für Deklarationen und Ablaufstrukturen

} restliches Kapitel

Syntax und lexikalische Einheiten

Syntaxbeschreibung für Aufbausyntax durch EBNF

identifier ::= nondigit { nondigit | digit }
nondigit ::= a | b | ... | z | A | B | ... | Z | _
digit ::= 0 | 1 | 2 | ... | 9

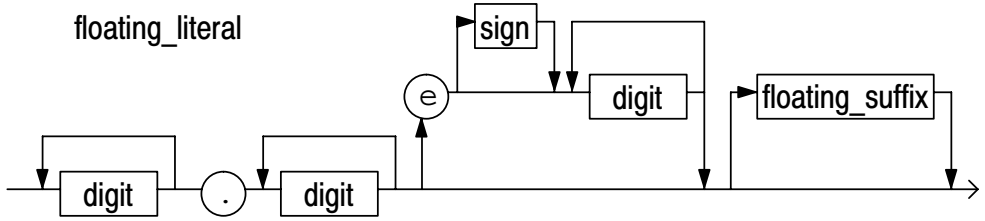
```
AnzahlDerZeilen
Anzahl_der_Zeilen   } zulässig
1_Rang
A#B                 } unzulässig
```

floating_literal ::= {digit}$^+$. {digit}$^+$[e[sign]{digit}$^+$] [floating_suffix]

```
1.0e-5f
3.1415              } zulässig
1. e-10
.347_8              } unzulässig
```

Begriffe: nichtterminale Zeichen identifier
terminale Zeichen wie . oder A
::= "ist erklärt gemäß"
Metazeichen | , { , } , }$^+$, [,] , (,)
Gestaltungshilfsmittel bei Syntaxfestlegung
Hintereinanderschreibung
Option [...]
Wiederholung { ... } bzw. { ... }$^+$
Gruppierung (...)
Fallunterscheidung ... | ...
Rekursion

Syntaxdiagramme (äquivalent zu EBNF)



Lexikalische Einheiten im Eingangsbeispiel

```
#include <iostream.h>
int main()
{  int summe, a, b;
   // Lies die Zahlen a und b ein
   cout << "a und b eingeben:";
   cin >> a >> b;

   /* Berechne die Summe beider
       Zahlen */
   summe = a+b;

   // Zeige das Ergebnis auf dem
   // Bildschirm an
   cout << "Summe=" << summe << endl;
   return 0;
}
```

Struktur der EBNF von C++

- Schichten (lexikalische Syntax – kontextfreie Syntax)
- Komplexe: Ausdrücke, Deklarationen, Anweisungen, Programmstruktur

→ siehe Anhang

Syntax

- lexikalische
- kontextfreie (Aufbausyntax) } für C++, → Anhang I
- kontextsensitive

kontextsensitive Syntax (umgangssprachlich,→ Anhang I)

Beispiele:

- Jeder Bezeichner muß deklariert sein
- Die Parameter bei Aufruf und Deklaration einer Prozedur müssen übereinstimmen
- Die Typen zweier Operanden zu einem Operator müssen typverträglich sein
- ...

Lexikalische Einheiten im Einzelnen

– Bezeichner

– Wortsymbole (Schlüsselwörter) [siehe Anhang I]

– Begrenzer

Abschluß	;	(Anweisung)
Anfang/Ende	{ .. }	(Block)
Abtrennung	,	(Aktualparameter)

– Kommentare

– Literale (s.u.)

- ganzzahlige
- reelle
- Zeichenliterale
- Zeichenkettenliterale

Programm–Layout als Pragmatik–Aspekt

– Programme lesbar für Entwickler (nicht Compiler)
  bei Entwicklung
  bei Veränderung
– Lesbarkeit durch Formatierungsregeln
  durch Leerzeilen, Leerzeichen, Einrücken
  durch entsprechende Bezeichner
  durch Kommentare
– vergleiche Layout und Grammatik
– Lesbarkeit hängt nicht nur vom Layout ab:
  Tricks
  Mißachtung von Methodik
  komplizierte Algorithmen

Einfache (skalare) Datentypen und ihre Operatoren

(zuerst vordefinierte, dann selbstdefinierte)

Ganzzahlige Datentypen:

Vordefinierte Datentypen

`short`	(üblicherweise 16 Bit)	–32768...32767
`int`	(16 oder 32 Bit)	s.o. oder –2147483648...2147483647
`long`	(32 oder 64 Bit)	entsprechend

`unsigned` (ohne Vorzeichen) – dadurch Wertebereich anders,
z.B.: `unsigned short` 0...65535

Über– oder Unterschreitungen werden zur Laufzeit in der Regel nicht gemeldet!

Ganzzahlige Literale

Oktalliterale	`0377`	`// (377)`$_8$ `= (255)`$_{10}$
Hexadezimalliterale	`0xAFFE`	`// (AFFE)`$_{16}$ `= (45054)`$_{10}$
oder	`0XAFFE`	
Dezimalliterale	`123`	

Suffix bei Literalen: `U,u` für unsigned
`L,l` für long

Konstantendeklarationen (compilezeitbestimmt)

```
int const KILO = 1000;
ind const MEGA = KILO * KILO;
```

Ganzzahlige Operatoren

Arithmetische Operatoren:		
`+`	`+i`	unäres Plus (wird weggelassen)
`-`	`-i`	unäres Minus
`++`	`++i`	vorherige Inkrementierung
	`i++`	nachfolgende Inkrementierung
`--`	`--i`	vorherige Dekrementierung
	`i--`	nachfolgende Dekrementierung
`+`	`i + 2`	binäres Plus
`-`	`i - 5`	binäres Minus
`*`	`5 * i`	Multiplikation
`/`	`i / 6`	Division
`%`	`i % 4`	Modulo (= Rest mit Vorzeichen von i)
Relationale Operatoren:		
`<`	`i < j`	kleiner
`>`	`i > j`	größer
`<=`	`i <= j`	kleiner gleich
`>=`	`i >= j`	größer gleich
`==`	`i == j`	gleich
`!=`	`i != j`	ungleich
Bit–Operatoren		
`<<`	`i << 2`	Linksschieben (Multiplikation mit 2er–Potenzen)
`>>`	`i >> 3`	Rechtsschieben (Division durch 2er–Potenzen)
`&`	`i & 7`	bitweises UND
`^`	`i ^ 7`	bitweises XOR
`|`	`i | 7`	bitweises ODER
`~`	`~i`	bitweise Negation
`&&`	`i && b`	logisches UND
`||`	`i || b`	logisches ODER
`!`	`!i`	logische Negation
Kurzformen bei Zuweisung		
`+=`	`i += 3`	`i = i + 3`
`-=`	`i -= 3`	`i = i - 3`
`*=`	`i *= 3`	`i = i * 3`
`/=`	`i /= 3`	`i = i / 3`
`%=`	`i %= 3`	`i = i % 3`
`<<=`	`i <<= 3`	`i = i << 3`
`>>=`	`i >>= 3`	`i = i >> 3`
`&=`	`i &= 3`	`i = i & 3`
`^=`	`i ^= 3`	`i = i ^ 3`
`|=`	`i |= 3`	`i = i | 3`

Reelle Datentypen (Gleitpunkttypen)

vordefinierte Datentypen

`float`	(32 Bit)	7	Stellen hinter
`double`	(64 Bit)	15	Dezimalpunkt
`long double`	(80 Bit)	19	bei Normierung

Gleitpunktliterale

```
-123.789e6f      1.8E6  88.009 1e-03
  1.8L
```

Datentyp `double`

Konstanten (compilezeitbestimmt)

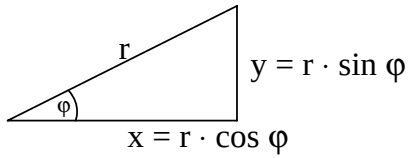
```
float const PI = 3.1415927;
float const PI_QUADRAT = PI * PI;
```

Operatoren

- arithmetische wie bei ganzzahligen bis auf `%`
- relationale wie bei ganzzahligen
- Kurzform bei Zuweisungen wie bei ganzzahligen

Beispiel für Ausdrücke mit vordefinierten Funktionen

Umrechnung von Polarkoordinaten in kartesische




```
#include <iostream.h>
#include <math.h>

double const PI = 3.141593;

void main() {
   double radius, winkel, x, y;

   /* Umrechnung von Polarkoordinaten
      in kartesische */

   cout << "Geben sie Radius in cm und "
        << "Winkel in Grad ein: ";
   cin >> radius >> winkel;

   x = radius * cos(PI*winkel/180.0);
   y = radius * sin(PI*winkel/180.0);

   cout << "Die kartesischen Koordinaten "
        << "in cm sind:" << endl;
   cout << "x=" << x << " y=" << y << endl;
}
```


Der Zeichentyp `char`

In einem Byte mit Vorzeichen durch `signed char` oder ohne Vorzeichen durch `unsigned char` dargestellt:

–128 ... 127 0 ... 255

ASCII – Zeichen 0 ... 127, Rest 128 ... 255 nicht standardisiert.

```
char const STERN = '*';
char zeichen; int wert;
zeichen = '*';
wert = int(zeichen);
            // Typkonversion →int: Stellenzahl
zeichen = char(wert);
            // wieder zugehoeriger char-Wert
```

char – Operatoren

Operator	Beispiel	Bedeutung
`<`	`d < f`	kleiner
`>`	`d > f`	größer
`<=`	`d <= f`	kleiner gleich
`>=`	`d >= f`	größer gleich
`==`	`d == f`	gleich
`!=`	`d != f`	ungleich

Zeichenliterale: `'z'` z ist darstellbares Zeichen

Darstellung von `\` , `'` , `"` durch `\\` , `\'` , `\"`

Darstellung nicht druckbarer Zeichen: `\o` bzw. `\xh`
wobei `o` bzw. `h` die Stellenzahl im ASCII–Zeichenformat in oktaler bzw. dezimaler Darstellung ist.

Der Datentyp `bool` (spezieller ganzzahliger Datentyp)

Literale:

`true` ≙ wahr
`false` ≙ falsch

```
bool gr_Buchst, kl_Buchst, Buchst;
char c;
cin >> c;
gr_Buchst = (c >= 'A') && (c <= 'Z');
kl_Buchst = (c >= 'a') && (c <= 'z');
Buchst = gr_Buchst || kl_Buchst;
cout << Buchst;
```

`bool` – Operatoren

Operator	Beispiel	Bedeutung
`!`	`!i`	logische Negation
`&&`	`a && b`	logisches UND
`||`	`a || b`	logisches ODER

Aufzählungstypen (intern ganzzahlig)

```
// Typdeklarationen
enum Betr_Zust {an, aus};
enum Erg_Zust  {richtig, eing_f, ber_f1,
                ber_f2, ausg_f};
enum Wochentag {mo, di, mi, dn, fr, sa, so};

// Objektdeklarationen
Wochentag Werktag = mo; //mit Initialisierung
Erg_Zust z1, z2;        //mehrere Variablen
enum {Start, Betrieb, Ende} Status;
                       //anonyme Typdeklaration

Werktag = mi;    später z.B. Werktag = dn;
Status = Start;  später Status = Ende;
z1 = ber_f1;
```

intern zugeordnete Werte 0, ..., # Aufz.typ – 1

bei Erg_Zust: 0, 1, 2, 3, 4; braucht uns i. a. nicht zu interessieren

für EA, für Sonderfälle Wertebelegung angebbar.

Alternativ für `Erg_Zust`:

```
enum Erg_Zust {richtig=1, eing_f=2,
               ber_f1=4, ber_f2=8,
               ausg_f=16};
```

Bemerkungen:

1) intern auf ganzzahlige Werte abgebildet;
2) anonyme Typdeklaration nur, wenn nur eine Variable benötigt wird;
3) ganzzahlige Wertzuordnung nur für EA bzw. wenn für effiziente Berechnung nötig. Aufzählungswerte für Indexbereiche von Feldern nutzbar, wird durch Zuordnung erschwert.

Ausdrücke skalarer Typen

Empfehlungen:

1) in C++ können mit `char`- oder `bool`-Werten und mit Aufzählungswerten als `int`-Werte gerechnet werden: nicht empfehlenswert!
2) Entscheiden Sie sich bei jeder Deklaration, welchen Basistyp sie verwenden:
   `short`, `int`, `long` und dabei `unsigned` oder normal für ganzzahlige Konstante oder Variable, `float`, `double`, `long double` für Gleitpunkt–Deklarationen.
3) Wenn Sie Ausdrücke bilden, achten Sie darauf, daß alle Operanden gleichen Typ haben; ebenso linke und rechte Seite einer Zuweisung.
4) In C++ gibt es viele implizite Typkonversionen innerhalb der ganzzahligen Typen, bzw. Gleitpunkttypen, aber auch zwischen beiden: nicht empfehlenswert!

Fallunterscheidungen

bisher Zuweisungen, ggf. in verkürzter Form wie `m+=1`, und Anweisungsfolgen

Betrachtung verschiedener Fälle

abhängig vom Wert eines Boole'schen Ausdrucks
(bedingte Anweisung, if-Anweisung)

abhängig von konkreten, explizit angegebenen Werten
(Auswahlanweisung, case-Anweisung, hier switch-Anw.)

Bedingte Anweisungen

```
if (schalter == an) {
   // Anweisungsfolge
};
```

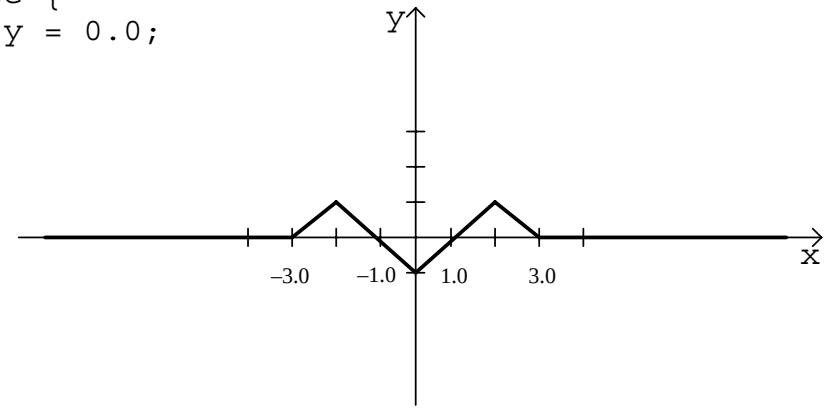
einseitig bedingte Anweisung

```
// Vorzeichenbestimmung
if (zahl >= 0) {
   sig = 1; }
else {
   sig = -1;
};
```

zweiseitig bedingte Anweisung


```
// Berechnung des Ordinatenwerts einer
// stückweise zusammengesetzten Funktion:

if (-3.0<=x && x<-2.0) {
   y = x+3.0;}
else if (-2.0<=x && x<0.0) {
   y = -1.0*x-1.0; }
else if (0.0<=x && x<2.0) {
   y = x-1.0; }
else if (2.0<=x && x<3.0) {
   y = -1.0*x+3.0;}
else {
   y = 0.0;
};
```

allg. bedingte Anweisung simuliert → Aufgaben



Fallstricke:

- Dangling-else-Problem
- falsches Semikolon
- zusammengesetzte Bedingungen

Ratschlag:

stets Block für Anweisungsfolge: then-Teil, else-Teil, elsif-Teile
vermeidet dangling else und erleichtert Einfügen weiterer Anweisungen

Auswahlanweisung

Bsp: jeder Wert ein Fall

```
int wert;
char eingabe;
cin >> eingabe;
switch (eingabe) {
   case 'I' : wert =    1; break;
   case 'V' : wert =    5; break;
   case 'X' : wert =   10; break;
   case 'L' : wert =   50; break;
   case 'C' : wert =  100; break;
   case 'D' : wert =  500; break;
   case 'M' : wert = 1000; break;
   default : cout <<
          "Ist keine roemische Ziffer!";
};
```

gleiche Behandlung für unterschiedliche Fälle:

```
switch Tag {
   case mo : Wochenanfangsbilanz(); break;
   case di : case mi: case dn :
   case fr : normale_Tagesbilanz(); break;
   case sa : Wochenabschlußbilanz(); break;
   default : Fehlerabbruch();
             // Fall wurde uebersehen
};
```

Bemerkungen:

- break-Anweisung verläßt die gesamte Anweisung
- es gibt keine Auswahllisten wie in anderen Sprachen
  di, mi, dn, fr oder di .. fr ; Simulation
- Werte in allen Fällen paarweise verschieden
- break-Anweisung nicht vergessen, sonst wird bei nächster Alternative weitergemacht
- Viele Fälle: besser bedingte Anweisungen anstelle von Auswahlanweisungen. Tritt in C nicht auf, da nicht bequem hinzuschreiben

Schleifen (Iterationen, Wiederholungen)

wiederholte Ausführung einer Anweisungsfolge

Unterscheidung: Anzahl der Fälle bekannt von 1 .. 100:
Zählschleife, for-Schleife

Anzahl der Fälle unbekannt :
while- oder until-Schleife, beides bedingte Schleifen

Keine Bedingung angegeben: Endlosschleife

Zählschleifen

$$fak(n) = \prod_{i=1}^{n} i$$

```
int nfak, n = ...;
// n - Fakultät für ein n, iterativ

// aufsteigend berechnet; n >= 1
nfak = 1;
for (int i=1; i<=n; i++) {
   nfak = nfak*i; // abg.: nfak *= i;
}

// ueblich absteigend; wieder n >= 1
nfak = 1;
for (int i=n; i>=1; i--)
   nfak = nfak*i; // abg.: nfak *= i;
}
```


```
// Fakultaetswerte von 1 bis 20
for (int n=1 ; n<=20; n++) {
   nfak = 1;
   for (int i=n; i>=1; i--) {
      nfak = nfak*i; // abg.: nfak *= i;
   }
   // Ausgabe der Fakultaetswerte
}
```

Bem.:

- for-Schleife in C++ auch anderweitig nutzbar: Semantik
  for (Initialisierung; Bedingung; Ausdruck) { Rumpf }:

  ```
  {Initialisierung
    while (Bedingung) {
      Schleifenrumpf;
      Ausdruck; // für Inkrem./Dekrem.
    }
  }
  ```

- Zählschleife kein Terminationsproblem; bei C++ aufpassen!
- for Schleife von C nur als Zählschleife verwenden
- insbes. keine Veränderung der Schleifenvariablen im Rumpf der Schleife!

Bedingte Schleifen

1. Problem: Summe von mehreren Werten ($\neq 0$), jetzt von außen eingegeben, “Länge” der Eingabe nicht vorbestimmt, 0 sei Ende der Eingabe.

while - Schleife:

```
...
int summe = 0, eingabe;
cin >> eingabe;

while (eingabe!=0) {  // Fortsetzungsbedingung
    // ist Negation einer Abbruchbedingung
   summe += eingabe;
   cin >> eingabe;
}
// Ausgabe
```

alternativ mit `do-while`-Schleife, wenn mindestens ein Wert $\neq 0$

```
do {  // keine until-Schleife; Pruefung
      // einer Fortsetzungbedingung am Ende
   cin >> eingabe;
   summe += eingabe; }
while (eingabe != 0);
// Ausgabe
```


2. Problem: Berechnung bis bestimmte Genauigkeit erreicht wird

```
float wurzel = 1.0, x;
float const eps = 1.0e-4;
cin >> x;
while (abs(wurzel*wurzel - x) > eps) {
   wurzel = 0.5*(wurzel + x/wurzel);
};
```

Bem.:
- üblicherweise while Schleifen, until-Schleifen
- until-Schleifen mit Abbruchbedingung simuliert durch:
  ```
  do { Rumpf }
  while (!Abbruchbedingung);
  ```
- Terminationsproblem!

Endlosschleifen

```
int summe = 0, eingabe;
while (true) {
   cin >> eingabe;
   if (eingabe==0) {
      break;
   }
   summe += eingabe;
}
```

Kontrollierte Sprünge

Sprunganweisungen:

```
break;// zum Verlassen der unmittelbar
      // umschließenden Programmeinheit,
      // z.B.Schleifen,Auswahlanweisungen
continue; // innerhalb Schleifen, sofort
      // nächster Durchlauf
return [expression]; // bei Unterprogrammen
      // sowie Hauptprogramm zur
      // Rückgabe eines Wertes (oder
      // Zustandsinformation)
goto label_identifier; // Sprung zu Sprungmarke
      // innerhalb eines Unter-
      // oder Hauptprogramms
```

Beispiele:

`break` für Auswahlanweisung, für Verlassen von (Endlos)Schleifen

`continue` entspricht exit zum Ende Schleifenrumpf; Beispiel später

`return` (→ nächster Abschnitt); ferner für Rückgabe eines Return-Parameters

`goto` (→ Aufgaben)

Bem.:

- strukturierte Programmierung: Sequenz, Iteration, Fallunterscheidung, keine Sprünge (goto-Kontroverse)
- saubere Sprünge: Sprungmarken sind Anfangs- oder Endpunkte von Kontrollstrukturen;
  bei `break`, `continue` und `return` der Fall

  strukturierte Prgogrammierung mit sauberen Sprüngen
- goto sorgfältig anwenden:
  (a) für Simulation von Kontrollstrukturen
  (b) für Effizienzverbesserung für Programme mit sauberen Sprüngen
  (c) für Programmstücke mit anderweitig klarer Struktur

Felder (Reihungen, Arrays)

Eingangsbeispiel Summenberechnung zweier Werte,
jetzt von 10 `float`-Werten: Deklaration eines Feldes,
Zusammenfassung von Werten des gleichen Typs

Feldobjektdeklaration, Feldtypdeklaration

```
unsigned const dimVal = 10;
float werte[dimVal]; // ein Feldobjekt
//Feldelemente werte[0],werte[1],...,werte[9]
//erstes Element hat Indexwert 0, letztes
//dimVal-1;
```

Oben implizite oder anonyme Felddeklaration. Nur falls genau ein Feldobjekt benötigt wird, oder diese Strukturen nicht anderweitig verwendet werden.
Besser mit expliziter Typdeklaration:

```
typedef float Feldtyp[dimVal];//Typdeklaration
```

Typdefinition Feldtyp

```
Feldtyp werte;  // Objekdeklaration des Typs
                // Feldtyp


typedef enum tage
            {mo, di, mi, dn, fr, sa, so};
typedef int StdProTag[so-mo+1];
```


Initialisierung von Feldern:

über Feldaggregate

```
Feldtyp werte = {1.0,2.0,3.0,4.0,5.0,6.0,
        7.0,8.0,9.0,10.0};
  // in Objektdeklaration,
  // falls Werte nicht veraendert werden:
Feldtyp const werte = ... (wie oben)
  // {} alle Komponenten von werte mit 0.0
  // initialisiert
```

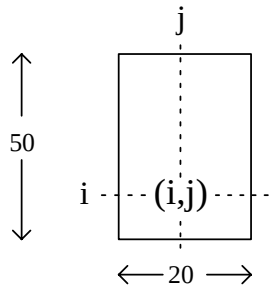
oder über Zählschleifen
bei großer Komponentenzahl, komplexer Berechnung der Komponentenwerte:

```
for (int i=0; i<dimVal; i++) {
   werte[i] = float(i*i); // Indizierung,
                          // Komp. Zugriff
}
```

Felder und Zählschleifen für Berechnung:
(Initialisierung (s.o.))

```
unsigned const dimVal=10;
typedef float f_A_T[dimVal];
f_A_T Werte;
float Summe=0, Mittelwert;
// Zaehlschleife z. Festlegen der Werte,
// evtl. über Eingabe
...
// Berechnung des Mittelwertes:
for (int i=0; i<dimVal; i++) {
   Summe += Werte[i];
};
Mittelwert = Summe/dimVal;
// Ausgabe ...
```

Mehrdimensionale Felder




```
double Mat[50][20]; // Feldobjekt,
  // 20x50-Matrix mit double-Komponenten;
  // ist eindim. Feld aus eindim. Feldkomp.
unsigned const dim1Val=50, dim2Val=20;
typedef double dZeilenT[dim2Val];
  // Zeilenvektortyp
typedef dZeilenT dMat_50_20_T[dim1Val];
  // Vektor aus Zeilen
dMat_50_20_T Mat;


// Bestimmung der minimalen Feldkomponente
// von Mat über 2 geschachtelte Schleifen:

double min = Mat[0][0];
for (int i=0; i<dim1Val; i++) {
   for (int j=0; j<dim2Val; j++) {
      double const Mij=Mat[i][j];
        // entspricht (Mat[i])[j]
      if (Mij<min) { min = Mij; };
   };
};
```


Zeichenfelder

Zeichenketten werden in C als Zeichenfeld mit '\0' (Terminator) als Endkennung abgespeichert. Wir besprechen zunächst statische Zeichenfelder. (Dynamische Zeichenketten → Zeigerabschnitt)

```
// ZK, ZF sind Zeichenfelder konst. Laenge
char ZK[] = "Z_Kette"; // rechts ZK-Literal,
                  // obere Grenze aus Literal
char const ZF[8] = "Z_Kette";
  // fuer Literal auch gleichbedeutend
  // {'Z','_','K','e','t','t','e','\0'}
```

Bemerkungen:

- statische und dynamische Felder: statisch d.h. Anzahl der Komponenten zur Compilezeit bekannt; dynamisch in Verbindung mit Zeigern
- mehrdimensionale Felder mittels Komposition aus eindimensionalen (Matrix: Vektor von Zeilen); Ablage im Datenspeicher: Zeilen hintereinander.
- keine Überprüfung der Indexgrenzen bei Zugriff, keine Zuweisung ganzer Felder
- Begriffe: Feld, Feldkomponente, Feldtypdefinition, Feldtypdeklaration, Feldobjektdeklaration, Zugriff (Indizierung) auf Feldkomponente, Aggregate, Initialisierung eines Feldobjekts, statische/dynamische, eindimensionale/mehrdimensionale Felder, Zeichenfelder, ZK-Literal, Aggregat für Zeichenfeld

Verbunde

Name: Verbund, Record, Strukturen, etc.

Zusammenfassung unterschiedlicher Komponenten

Typdeklaration, Objektdeklaration einfacher Verbunde

```
enum FarbT {rot, gelb, blau};
struct PunktT{                  // Typ PunktT;
   float x, y; // koordinaten;  Typdefinition
   FarbT farbe;
   bool sichtbar;
   bool in_Ausschnitt;
} pkt1;                    // Objektdeklaration

// besser getrennt
struct PunktT {       // Typdeklaration
   s.o.
};
PunktT pkt1, pkt2;    // Objektdeklarationen
```

Zugriff auf Komponenten

```
pkt1.x = 12.0; pkt1.y = 15.0;
pkt1.farbe = rot; pkt1.sichtbar = true;
if((x1<=pkt1.x && pkt1.x<=x2)&&
   (y1<=pkt1.y && pkt1.y<=y1)) {
   pkt1.in_Ausschnitt = true;}
else {
   pkt1.in_Ausschnitt = false;
};
```

Anweisungsfolge für die Verarbeitung der (einiger) Komponenten

```
pkt2.y = pkt1.y; pkt2.x = pkt1.x;
             //gleiche Koordinaten für pkt2
```

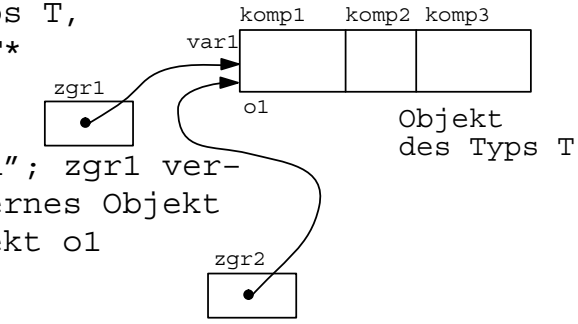

Zeiger und Referenzen

Bisher Objekte über Bezeichner eingeführt,
Objekte von bel. zusammengesetztem Typ,
unauflösliche Bindung Bezeichner, interner Wert (an best. Adresse).

Jetzt Zeiger (Verweis, Pointer, Zugriff) auf Objekte, die nacheinander auf verschiedene Objekte des gleichen Typs verweisen können.

Zeigerdeklarationen

```
T var1 = ..., var2;
     // Initialisierung, T sei z.B. Verbund
T *zgr1; // Verweis auf
// Objekt des Typs T,
// zgr1 hat Typ T*

zgr1 = &var1;
// & "Adresse von"; zgr1 ver-
// weist auf internes Objekt
// von var1, objekt o1

T *zgr2 = &var1;
// zgr2 verweist auf das gleiche
// Objekt o1
```



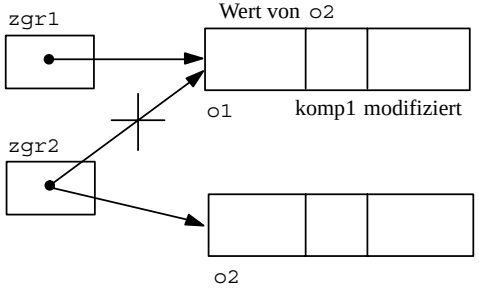
oder alternativ

```
T *zgr2 = ptr1;
// zgr2 erhält Zeigerwert von zgr1, beide
// verweisen auf das gleiche Objekt o1
```

Zeiger sind typisiert (`T*`) und können nicht auf Objekte eines anderen Typs verweisen

Anstatt der letzten initialisierten Deklaration hätten wir schreiben können:

```
T *zgr2;
zgr2 = zgr1;
// Zuweisung d. Zeigerwerts von zgr1 an zgr2


var2.komp1=...;var2.komp2=...;var2.komp3=...
// Objekt var2 (o2) erhaelt (neuen) Wert;
// bisher verweisen zgr1 und zgr2 auf das
// Objekt o1


*zgr1 = var2; // zgr1 verweist auf Objekt o1
// Dereferenzierung: Uebergang von Zeiger zu
// angezeigtem Objekt; dieses Objekt erhält
// neuen Wert (von o2)
*zgr1.komp1 = ...;
// komp1 von o1 erhaelt neuen Wert

// oder kuerzer fuer *zgr1.komp1 bei
// Veraenderung einzelner Komponenten:
zgr1->komp1 = ...;

zgr2 = &var2;
// zgr2 verweist
// jetzt auf o2
```



alternativ für letzteres, da `var1` und `var2` im Speicher hintereinander liegen:

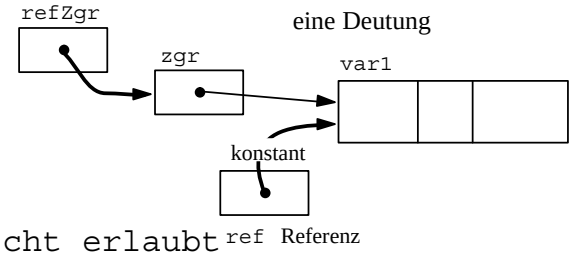
```
zgr2 = zgr1 + sizeof(var1);
       // gefaehrlich und funktioniert nicht
```

Bemerkungen:

- nicht `T* zgr1, zgr2;` schreiben; `zgr1` ist vom Typ `T*`, `zgr2` vom Typ `T`
- bisher Verweise auf Objekte im Laufzeitkeller, später auf Halde
- bei `T const *zgr1 = &var1;` ist `zgr1` ein Zeiger auf ein konstantes Objekt, der Zeiger ist nicht konstant; verschiedene Deklarationen verwenden

Referenzen

```
T var1;
T var2;
T *zgr = &var1; // Zeiger auf var1
T &ref = var1;  // ref ist Referenz auf var1
// ref ist neuer Name (Alias) für var1; wird
// bei Deklaration gesetzt und nicht mehr
// verändert

var1.komp1 = ...;
zgr->komp1 = ...;
ref.komp1  = ...;

ref = &var2; //nicht erlaubt
T *&refZgr = zgr; // Referenz auf Zeiger
ref = var2;  // Referenz bleibt gleich
             // Objekt wird verändert;
             // Wirkung von var1 = var2
```



Referenzen zum Abkürzen von Zugriffspfaden:

```
K11T &Komp = var1.komp1.komp11; // K11T, K2T
K2T &bestKomp = F[251]; // entspr. dekl. Typen
```

Bemerkungen:

- Zeiger auf Referenzen sowie Referenzen auf Referenzen unzulässig, Referenzen auf Zeiger wohl
- für alternativen Namen auf Variable, Konstante, Funktion (→ später)

Unterprogramme

Funktionen:	liefern Berechnungswert als Ergebnis; werden in Ausdruck aufgerufen
(Verfahrens-) Prozeduren:	liefern komplexere Berechnung haben Charakter einer komplexen Anweisung, Wert des Typs `void` (undef.)

Funktionen

Funktionsspezifikation
(Funktionskopf, in C Prototyp genannt)

Syntax:

```
Rueckgabetyp Funktionsbez(Formalparameterliste);
```

```
int strlen(char const *string);
                    // Funktionsprototyp, mit
                    // Formalparameterliste
void main() {
   char *str = "Eine Zeichenkette";

   cout << str << " hat die Länge: "
        << strlen(str) << endl; // Aufruf
}
```

Formen der Parameterliste einer Funktionsspezifikation

leere Liste: `int func();`
gleichwertig ist: `int func(void);`
Liste mit Parametertypen: `int func(int, char);`
Liste mit Parametertypen und –bezeichner:
`int func(int x, char y);`

Empfehlung: mit Parameterbezeichnern

Funktionsrumpf

andere Namen: Funktionsdeklaration, –implementation

Wiederholung der Schnittstelle + Implementation
(Berechnungsvorschrift)

Syntax:
`Rueckgabetyp Funktionsbez(`**Formalparameterliste**`)` **block**

Die Formalparameterliste muß für jeden Parameter einen Namen einführen

```
// Funktionsimplementation
int strlen(char const *x) {
   int i = 0; char const terminator = 0;
   while (x[i]!=terminator) {
      i++
   };
   return i;
}
```

Funktionsaufruf

Syntax:
`Funktionsbez(`**Aktualparameterliste**`)`

```
void main() {
   ... // s.o.
   cout << strlen(str);
            // Ausgabe eines Wertes
}
```


Vordefinierte Parameter

Vorgabe- / Vorbesetzungsparameter (engl. defaults) nach den anderen Parametern.

Beispiel:

`hex` konvertiere `long`(`int`)-Zahlen in Zeichenkette aus Hexadezimalzahlen. Im zweiten Parameter bedeutet 0:
Zeichenkette gerade so lang, wie benötigt;
sonst wie vom Aufrufer angegeben.

```
extern char* hex(long, int = 0);

cout << "**" << hex(31) << hex(32,3) << "**";
```

wird interpretiert als:

```
cout <<"**"<< hex(31,0) << hex(32,3) << "**";
```

und beide liefern das gleiche:

```
**1f 20**
```

Prozeduren

Prozedur = komplexere Berechnung; Aufruf ist Anweisung

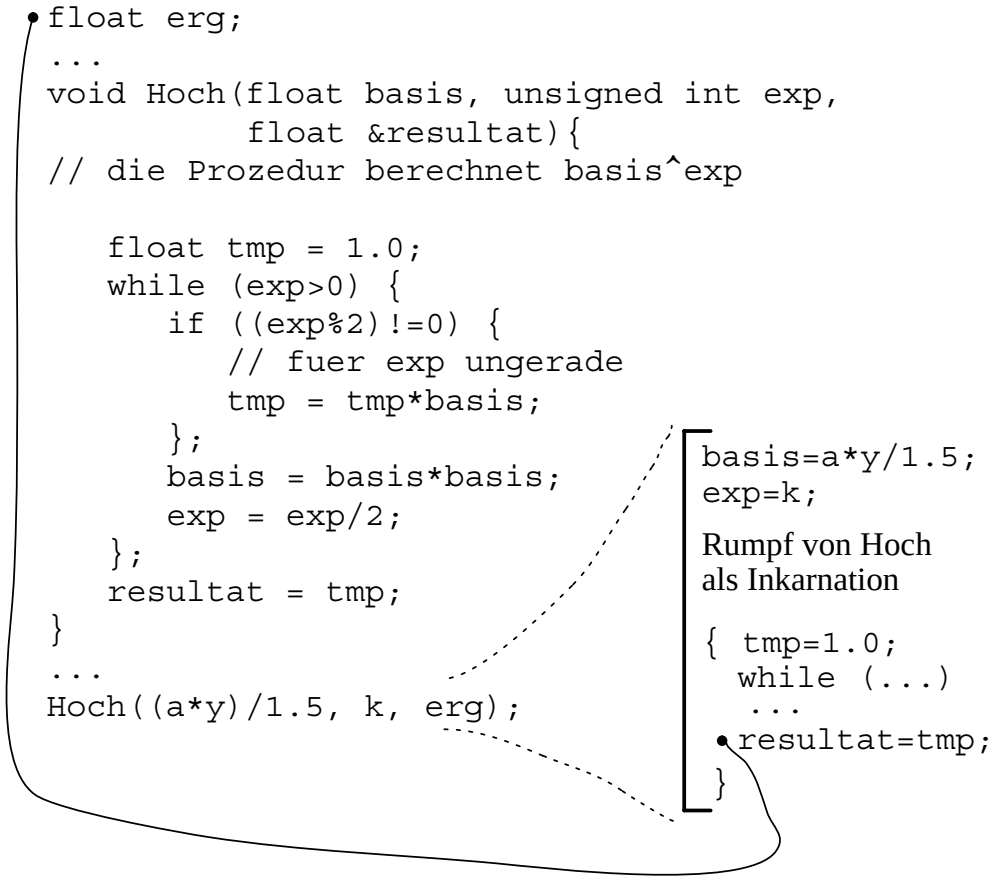
```
float erg;
...
void Hoch(float basis, unsigned int exp,
          float &resultat){    // Deklaration;
   // Die Prozedur berechnet basis^exp
   // basis, exp  Eingangsparameter,
   // resultat    Ausgabeparameter

   float tmp = 1.0;
   while (exp>0) {
      if ((exp%2)!=0) {
         // fuer exp ungerade
         tmp = tmp*basis;
      }
      basis = basis*basis; exp = exp/2;
   };
   resultat = tmp;
};
...
Hoch((a*y)/1.5, k, erg); // Aufruf
   // a,y als float dekl., k als unsigned

...
void exchange (float &x, float &y) {
   // Vertauschung zweier Werte;
   // beide Parameter sind Ein-/Ausgabeparam.
   float zwischen;
   zwischen = x; x = y; y = zwischen;
}
...
exchange(a,b); // Aufruf
```


Bedeutung eines Unterprogrammaufrufs und Parameterübergabemechanismen

Übergabemechanismen:

call-by-value: (Aufruf über den Wert)	Eingabeparameter wird in Formalparameter kopiert
call-by-reference (Aufruf über die Adresse)	Aktualparameter wird genommen und verändert

Erklärung Bedeutung Prozeduraufruf



Aufruf mit Zeigern

- ist Aufruf über den Wert, d.h. der Zeiger wird nicht verändert (ggf. die lokale Kopie)
- Das angezeigte Objekt kann in der Prozedur natürlich geändert werden

Hauptprogramm

Funktion main

```
int main() { // int kann entfallen
   ...
   return 0;
}
```

Rückgabewert z.B. für gezielte Reaktion auf Fehler.

```
void main() {
   ...
} // braucht keine return-Anweisung
```

Felder und Unterprogramme

- Unterprogramm für verschiedene Komponentenzahl

```
...
unsigned const dimVal = 100;
typedef float f_FT[dimVal];
f_FT xv, yv;
...
float Skalarprodukt(f_FT avekt,f_FT bvekt){
   int og1, og2;
   float erg = 0.0;
   og1=int(sizeof(f_FT)/sizeof(float));
   og2=int(sizeof(f_FT)/sizeof(float));
   if(og1!=og2) { ... // Fehlerabbruch };
   for (int i=0; i<og1; i++) {
      erg = erg + avekt[i] * bvekt[i];
   }
   return erg;
}
```

- jetzige Formulierung noch vom Typ der Komponenten abhängig; vom Komponententyp nur verlangt, daß +, * verfügbar; später Nutzung von Generizität
- bei größeren Vektoren call-by-reference, obwohl Felder hier Eingabeparameter sind!

Zeiger und Unterprogramme

Parameter und Zeiger

```
unsigned const dimVal=10;
float erg;
typedef float f_FT[dimVal];
f_FT *pF1, *pF2;
  // pF1 und pF2 sind Zeiger auf Felder

float Skalarprodukt(f_FT *avekt,
                    f_FT *bvekt) {
  // Formalparameter jetzt Zeiger auf Felder
  // Ersetze in der letzten Formulierung
  // jeweils avekt bzw. bvekt durch
  // (*avekt), (*bvekt)
   ...
}

// Felder initialisieren über EA oder mit
// Schleifen
...

erg = Skalarprodukt(pF1, pF2); // Aufruf
```

```
typedef f_FT* Z_f_FT;
Z_f_FT pF1, pF2;
```


übliche, unsaubere Lösung in C

```
unsigned dimVal=10; float erg;
typedef float *Z_f;
typedef float f_FT[dimVal];
f_FT F1, F2; // F1, F2 sind Zeiger auf die
             // erste Komponente
Z_f Feldanker1=F1; Feldanker2=F2;
  // Feldanker1, Feldanker2 zeigen je auf
  // erste Komponente und implizit auf
  // F1, F2

float Skalarprodukt(Z_f avekt, Z_f bvekt,
            unsigned len1, unsigned len2) {
  // Formalparameter jetzt Zeiger auf
  // Anfangskomponente; len1, len2 Laenge
  // der zu uebergebenden Felder

  if (len1 != len2) { ...
    // Fehlerabbruch; keine Garantie, dass
    // Felder gleiche Laenge haben
  }
  // ab hier wieder alte Formulierung

};
...

erg=Skalarprodukt(F1, F2, dimVal, dimVal);
       // Aufruf
```

Nutzung auch für dynamische Zeichenketten (zur Laufzeit variabel): wird hier nicht besprochen

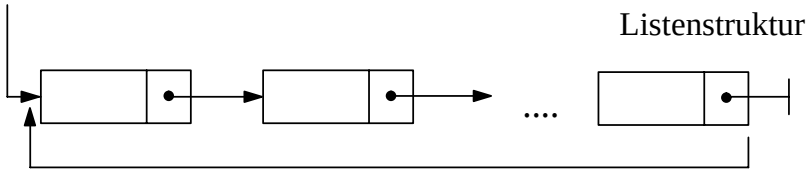
Zeiger auf Unterprogramme

```
#include <iostream.h>
int max(int x, int y) {
   if (x > y) { return x; }
   else       { return y; };
};
int min(int x, int y) {
   if (x < y) { return x; }
   else       { return y; };
}; //beide Funktionen haben gleiches Profil

int main() {
   int zahl1=10, zahl2=20; char eingabe;
   int (*fp)(int, int); // fp Zeiger auf
      // Funktion mit bestimmtem Profil
   while (true) {
      cout << "s=min, g=max, e=Ende:\n";
      cin >> eingabe;
      switch (eingabe) {
         case 's' : {fp = min; break;};
           // Zuweisung von min
         case 'g' : {fp = max; break;};
           // Zuweisung von max
         case 'e' : {goto Ende;};
         default: {cout<<"Falsche Eingabe";
                   continue;};
      };
  // Dereferenzierung des Funktionszeigers
  // und Aufruf der entsprechenden Funktion
      cout << (*fp)(zahl1,zahl2) << endl;
   };
   Ende: return 0;
}
```


Haldenobjekte, Listenverarbeitung

Zeiger und Haldenobjekte

- bisher Objekte beliebig komplizierter Struktur mit Typkonstruktor Feld, Verbund etc;
  in ihrer Struktur in Objektdeklaration festgelegt:
  "statische" Objekte
- Notwendigkeit dynamischer Objekte:



Anzahl nicht vorherbestimmbar zur Programmerstellungszeit, ändert sich zur Laufzeit

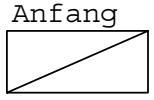
- bisher: Objektfestlegung durch Objektdeklaration
  unauflösliche Bindung externer Name – interner Wert
  Erzeugung des Objektes durch Abarbeitung einer Objektdeklaration

  jetzt: Objekterzeugung im Anweisungsteil
  Bereich auf den abgelegt wird: Halde (Heap)
  entsprechende Objekte: Haldenobjekte
- Haldenobjekte:
  durch Zeiger benannt (implizite, erzeugte Bezeichnung)
  durch Zeiger untereinander verbunden (verkettet, verzeigert)

Deklaration der Typen von Haldenobjekten, Zeigern

Wir beschränken uns auf Verbunde, da Objekte auf Halde verkettet werden sollen; Zeiger auf einzelne Felder auf der Halde ebenfalls möglich.

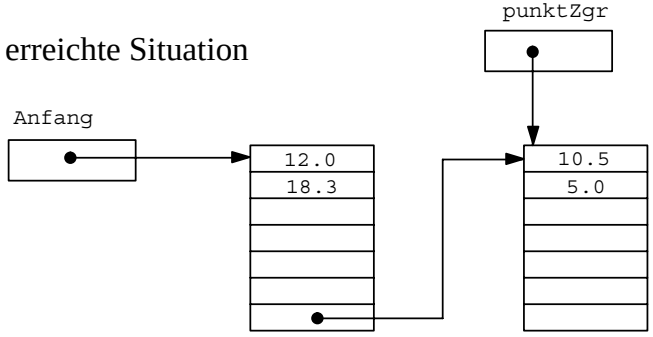
```
// Struktur der Haldenobjekte
void const *NULL=0;

struct PunktT;
typedef PunktT *zpunkt;
         // Zeiger auf Haldenobjekte
struct PunktT { // Struktur der Haldenobj.
   float x, y;
   farbT farbe;
   bool sichtbar;
   bool in_Ausschnitt;
   zpunkt n_koord; // Verw. auf n. Koordinate
   zpunkt next;  // Zeiger fuer Verkettung
};               // der Listenelemente

// Ankerzeiger
zpunkt Anfang=NULL; //zunaechst noch undef.
     // verweist spaeter auf Anfang der Liste
zpunkt punktZgr; // spaeter Zeiger auf akt.
                 // Element
```



Anlegen, Verändern, Verketten von Haldenobjekten

```
punktZgr = new PunktT;
           // 1. Element der Liste, Verweis
Anfang = punktZgr;
punktZgr->x = 12.0;
punktZgr->y = 18.3;
// andere Komponenten (farbe, etc.)
// noch undef.

punktZgr->next = new punkt;
 // erzeugt 2. Objekt und Verweis

punktZgr = punktZgr->next;

Anfang->next->x = 10.5;
 //Veraenderung ueber Zugriffspfad
punktZgr->y = 5.0 // Veraenderung ueber
                  // Laufzeiger
```

erreichte Situation

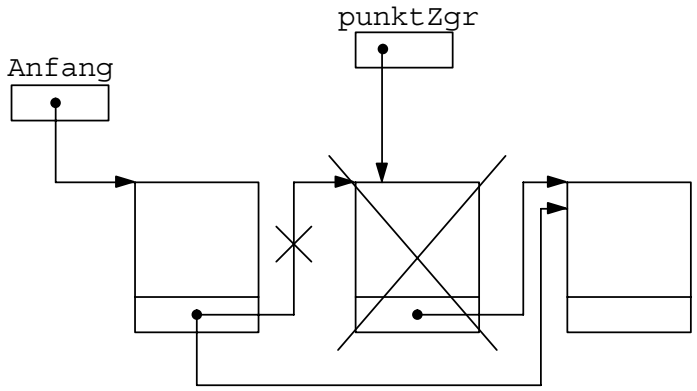


anonyme Haldenobjekte
Verweise bei Allokation jeweils mit erzeugt

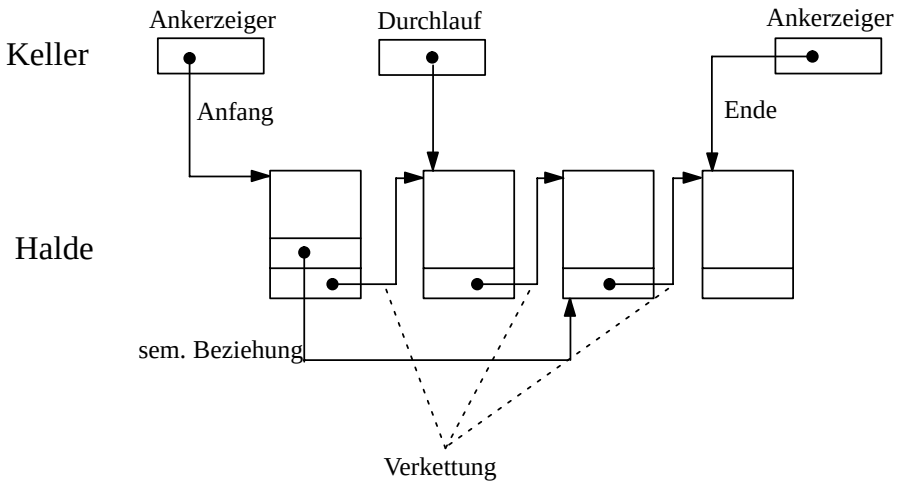
Erinnerung: auf Objekt (auf Halde oder Keller) können mehrere Zeiger verweisen.

Löschen von Elementen

```
// weiteres Listenelement

punktZgr->next = new PunktT;


// Loeschen des 2. Elements

punktZgr     = Anfang->next;
Anfang->next = punktZgr->next;
delete punktZgr;
```



Aufgaben von Zeigern



- Zeiger von außen:
  Einrichtung/Abkürzung eines Zugriffspfades
  Ankerzeiger
  Durchlaufzeiger
- Zeiger zwischen Objekten:
  Zusammenhang (Verkettung; z.B. für Durchlauf)
  inhaltliche (semantische) Beziehungen

Gefahren mit Zeigern

- unübersichtliche Strukturen
  (Zeiger sind gotos der Datenstrukturseite)
- Ansprechen/Verändern eines Objektes über verschiedene Namen (Aliasing):
  Veränderung eines Haldenobjekts über einen Zugriffspfad = Veränderung über anderen Zugriffspfad (Gefahr undurchsichtiger Programme)
- nicht mehr ansprechbares Haldenobjekt: ausgehängt (inaccessible object)
- hängender Zeiger nach `delete` (dangling references)

Effizienz und Zeiger

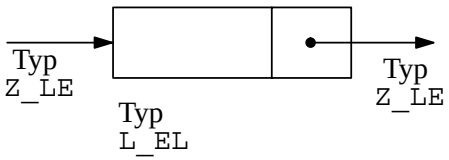
Speicherbereinigung (garbage collection)
wegen Zerstückelung der Halde:

- Kennzeichnung der freien Objekte
- Aufsammeln der freien Objekte
- Zusammenschieben des Haldenspeichers

Listenverarbeitung über Unterprogramme

einfache Deklarationen für lineare Liste

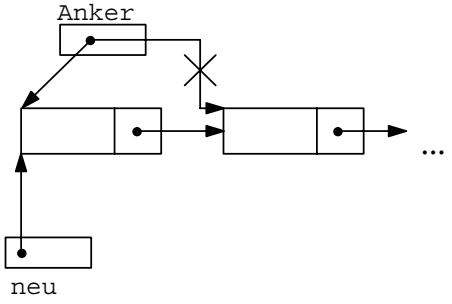
```
void const *NULL=0;
struct L_EL;
typedef L_EL *Z_LE;
struct L_EL {
   Info_T Info;
   //Info_T sei ein passend dekl. Typ
   Z_LE next;
};

Z_LE LinList;
```



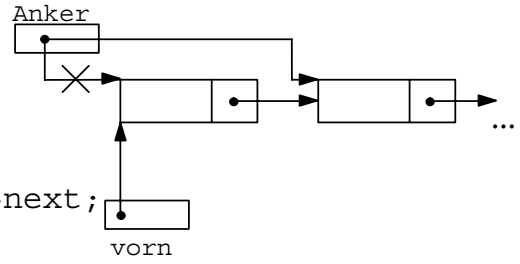
Strukturfestlegung

```
void LeereListe(Z_LE &Anker) {
   Anker = NULL;
};

unsigned LaengeDerListe(Z_LE ListenAnf) {
// Bestimme Laenge einer Teilliste
   unsigned Erg;

   if (ListenAnfang==NULL) {
      Erg=0; }
   else {
      Erg=1+LaengeDerListe(ListenAnf->next);
   };
   return Erg;
};
```

```
void AnfuegeVorn(Z_LE &Anker, Info_T i) {
// Fuegt vorne Listenelement ein;
// setzt Anker neu
   Z_LE neu;

   neu = new L_EL;
   neu->Info = i;
   neu->next = Anker;
   Anker = neu;
};
```




```
void LoescheVorn(Z_LE &Anker) {
// Loescht erstes Listenelement;
// setzt Anker neu

   Z_LE vorn;
   if (Anker!=NULL) {
      vorn = Anker;
      Anker = Anker->next;
      delete vorn; };
   else {
      ... // Fehlermeldung; Abbruch
   };
};
```




```
void Info_LE(Z_LE Anker, Info_T &i) {
   if (Anker!=NULL) {
      i=Anker->Info; };
   else {
      ... // Fehlermeldung; Abbruch
   };
};
```



```
// Verwendung der Liste

void main() {
   ...
   LeereListe(LinList);
   AnfuegeVorn(LinList, i1);
   AnfuegeVorn(LinList, i2);
   ...
   if (LaengeDerListe(LinList)!=0) {
      LoescheVorn(LinList);
   };
   ...
   if (LaengeDerListe(LinList)!=0) {
      Info_LE(LinList, i3);
   };
};
```

Weitere Operationen nötig: Navigieren, Einfügen innerhalb der Liste, Löschen innerhalb der Liste, etc.

Programmstruktur, Gültigkeit, Sichtbarkeit

Blockstruktur: lokale, globale Objekte

```
// Block 1:
{  unsigned i, j; // Dekl. 1,2

   // Block 1.1:
   {  unsigned i; // Dekl. 3
      float j;    // Dekl. 4

      j = ...; // Zuweisung an Objekt von
           // Dekl. 4: lok. Obj.;
           // Dekl. 2 verdeckt, aber gültig
   };
   ...
   i = ...; // Zuweisung an Obj. von Dekl.1
   j = ...; // Zuweisung an Obj. von Dekl.2
};
```

- globale Objekte, lokale Objekte
- Gültigkeitsbereich einer Deklaration:
  bis Ende Programmeinheit, hier Block
  außerhalb ungültig, Compiler prüft dies (Sicherheitsaspekt)
- Lebensdauer und Speicherverwaltung :
  außerhalb eines Blocks nicht gültig, nicht existent
  innerhalb eines Blocks ggfs. unsichtbar (verdeckt)
  nach Blockende Speicherbereich aufgegeben:
  Laufzeitkeller (effiziente Speicherverwaltung)

Unterprogramme: Gültigkeitsbereich, Existenzbereich

- Lokale Variablen einer Funktion/eines UPs außerhalb nicht gültig (wie bei Blöcken; keine Prozedurschachtelung in C)
  z.B. UP zum Vertauschen zweier Werte
- Formalparameter sind von Anzahl, Typ und Reihenfolge außerhalb sichtbar, die Formalparameternamen nicht
- Besonderheit:
  `static`-Variablen: Nur einmal auf statischem Speicherbereich angelegt, tauchen nicht in UP–Inkarnation auf

```
#include <iostream.h>

void func() {
     // zaehlt die Anzahl der Aufrufe
   static int anz = 0;

   anz++;
   cout << "Anzahl = " << anz << endl;
};

void main() {
   for (int i=0; i<6; i++) {
      func();
   };
}
```

Überladung von Funktionen

- Variablenbezeichner nicht überladbar
  gleiche Bezeichner = verschiedene Variablen, werden nicht durch Typ unterschieden
- Funktions– und Unterprogrammbezeichner überladbar:
  Verschiedene Deklarationen zum gleichen Funktions– oder Unterprogrammnamen:
  Überladung (overloading)
- Je nach Parametertypprofil (Anzahl, Reihenfolge, Typ der Parameter) wird vom Compiler das passende UP gewählt.

```
void exchange(double &x, double &y) {
   ....
};

void exchange(int &x, int &y) {
   ....
};


void main() {
   double dbl1 = 100.1, dbl2 = 33.3;
   int zahl1 = 5, zahl2 = 7;
   exchange(dbl1, dbl2);
   exchange(zahl1, zahl2);
}
```


Programmstruktur/Aufbau eines C-Programms

Compiler-Instruktionen (zum Beispiel `#include`)
Deklaration von globalen Variablen
Funktionsprototypen (Unterprogrammschnittstellen)

Implementierung, eigentliche Problemlösung:
`void main() {`
`  // Folge von Deklarationen und`
`  // Anweisungen };`

und Funktionsdefinitionen (Unterprogrammcode)

Beispiel:

```
#include <iostream.h>
float a=1.0, b=2.0;
  // globale Deklaration: Gefahr !
  // wie umschließender Block
void exchange(float &x, float &y);  // PT

void main() {
   // neuer Block
   float a = 10.0;
   //globales a gültig nicht sichtbar
   cout << "globales a =" << ::a << endl;
   // scope-Operator (::) erlaubt Zugriff
   // auf unsichtbares globales a
   {  int c=30; // neuer Block
   }
   cout << c << endl; // <- Compiler
   // liefert Fehler: c nicht gültig
   exchange(a,b);
};

void exchange(float &x, float &y) {...};
```

Resumee:

- lok. Deklarationen in Block, Unterprogramm:
  nahe an der Verwendung
  von außen nicht zugreifbar: Information Hiding
- globale Variable nicht (nur diszipliniert) anwenden
- Variablen werden verdeckt, Funktionen überladen
- static-Variable statisch
- scope-Operator erlaubt Zugriff auf globales evtl. verdecktes Objekt
- werden später andere und bessere Strukturierung eine C++–Programms kennenlernen!

Zusammenfassung

Anweisungen

- einfache Anweisungen
  Zuweisungen
  Prozeduraufrufe
  Sprunganweisungen
- Zusammenges. Anweisungen
  Anweisungsfolgen in Block
  Fallunterscheidungen
    bed. Anweisungen
    Auswahlanweisungen
  Schleifen
    Zählschleifen
    while-Schleifen
    ”until-Schleifen”
    Endlosschleifen

} Kontrollstrukturen (Vorschriften für Zusammenbau) für Ablaufkontrolle

Im Block oder Unterprogramm: Anweisungsteil
  später auch in anderen Bausteinen

statischer Programmtext soll dyn. Durchlauf möglichst entsprechen

C ist wertorientierte Sprache; haben wir nicht genutzt; führt zu Programmierung mit Seiteneffekten

Datenstrukturierung

- einfache skalare Datenobjekte mittels
  - ganzzahliger Datentypen
  - reeller Datentypen
  - `bool`
  - `char`
  - Aufzählungstypen
- Zusammengesetzte Datenobjekte durch
  - Felder
  - Verbunde
  - Zeiger auf/in Datenobj.
  - Referenzen auf Datenobj.

  } Datentypkonstruktoren: Vorschriften für Zusammenbau zusammeng. Datentypen

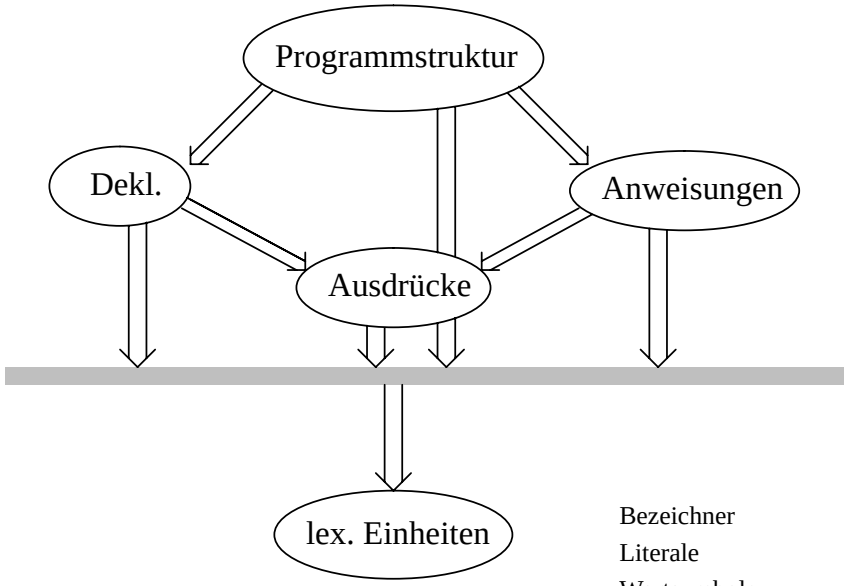
- Klassifikationen (orthogonal):

  einfache DTen — vordef. DTen

  zusammeng. DTen — selbstdef. DTen

- weitere Ergänzung durch Klassen

Gruppierung der Konstrukte in der k.f. Grammatik



Bezeichner
Literale
Wortsymbole
...

Entsprechung von Kontroll- und Datenstrukturen

Konstruktionsmuster	Anweisungsform	Datentyp
atomares Element	Zuweisung	skalarer Typ
Zusammenfassung	Anweisungsfolge (Block)	Verbundtyp
Fallunterscheidung	bed. Anweisung	`bool`
Auswahl	Auswahlanweisung	diskreter Typ (Aufz.)
Wiederholung Anzahl bekannt	Zählschleife	Feldtyp
Wiederholung Anzahl unbekannt	while- oder repeat–Anweisung	“Sequenztyp”
Rekursion	rek. Prozedur	rek. Datentyp
Allg. Graph	Sprunganweisung	Listenstruktur

Entsprechung eng
(Zählschleife, Felder)
oder lose
(rek. Prozedur, rek. Datentyp)

3 Methodisches Programmieren im Kleinen und Suchen/Sortieren

Lernziele:

Vorgehen beim Erstellen kleiner Programme
Effizienzbetrachtung, Test, Dokumentation
Standardkenntnisse Sortieren, wenig über Suchen

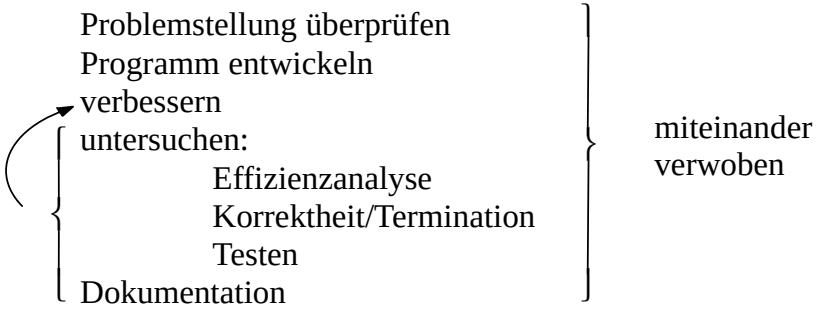
Die Entwicklung kleiner Programme

Vorgehen beim Lösen

– bisher:

Problemformulierung (meist gegeben: Aufgabe)
direkt Programm erstellt:
Hilfsmittel: schrittweise Verfeinerung
Wahl geeigneter Bezeichner
Verwendung von Kommentaren
dabei möglichst gute Gestaltung der Bedieneroberfläche

– systematisches Vorgehen bei kleinen Programmen



Eigenschaften der Lösung

– Problem → zugehöriges Programm
es gibt (unendl.) viele Lösungen

Lösungen: schwach, stark } äquivalent: gleiches Ergebnis; gleiche interne Zustandsübergänge

– welche Eigenschaft soll Lösung haben?

- Zuverlässigkeit
  Korrektheit: löst geg. Aufgabenstellung
  Robustheit gegen falsche Eingabedaten
  Ausfallsicherheit gegen Soft-/Hardwarefehler
- Benutzerfreundlichkeit: UI–Gestaltung
  Verständlichkeit
  Angemessenheit
  vernünftiges Fehlerverhalten
- Flexibilität
  Übertragbarkeit (Portabilität)
  Anpaßbarkeit (Veränderung, Erweiterung)
- Verständlichkeit des Programms
  Lesbarkeit
  Einfachheit
- Effizienz
  Laufzeit
  Speicherplatz
  wichtig sind auch Prozeßaspekte

Sortieren als Voraussetzung für effiziente Suche

Allgemeines

– häufig auftretendes Problem
  bei Übersetzung in Symboltabelle
  in Datenbanken bei großen Informationsbeständen
  in bel. Anwendungsprogrammen

– hier nur einfache Form
  in Feld im Hauptspeicher (internes Suchen,
    Random–Access Zugriffsstruktur)
  später bessere Möglichkeiten (→ Kap. 5)
  Suchen insb. auf kompl. Daten, oft auf ext. Speichermedien

– liegen die zu suchenden Datenelemente sortiert vor?
  nein: sequentielle Suche
  ja: binäre Suche

– zu suchende Elemente Suchinfo (mit Schlüssel)
  der Einfachheit halber Suchinfo nur Schlüssel
  setzen diesen `unsigned int`:
    könnte bel. Typ mit Vergleichsop. sein

– Motivation für Beschäftigung mit Sortieren in diesem Kapitel

Lineare (sequentielle) Suche

– Programm

```
enum zustT{gef, nicht_gef};
zustT status;
unsigned const dimVal=100; int Wert=... ;
typedef unsigned int itemT; // Aenderbarkeit
typedef int indexT;
indexT anzElem;
typedef itemT it_FT[dimVal];
it_FT feld;

// feld erhält Werte, z.B. über EA;
// anzElem ist die Anzahl der Elemente;
// Feldelemente: a[0], ..., a[anzElem-1]

void linSuche(it_FT &f, itemT ges_Wert,
              indexT ia, indexT ie,
              indexT &rsltInd, zustT &zust) {
  // besser it_FT &f,
  // d.h. call by reference
  // zu suchender Wert: ges_Wert;
  // rsltInd ist Index im Feld, bei Erfolg;
   indexT index, indexT og=ie+1;
   index = og; // Vorbes. ausserhalb
   for (indexT i=ia; i<og; i++) {
      if (f[i]==ges_Wert) {
         index=i; break;
      };
   };
   if (index < og) {
      rsltInd = index; zust = gef; }
   else {
      zust = nicht_gef;
   };
};
```

Aufruf:

```
...
// z.B. im Hauptprogramm
linSuche(feld, Wert, 0, anzElem-1,
         i, status);
if (status==gef) {
   cout << "Wert " << Wert << " an Stelle "
        << i << " gefunden." << endl; }
else {
   cout << "Wert nicht gefunden." << endl;
};
```

– Analyse

Sei n die Anzahl der Komponenten:

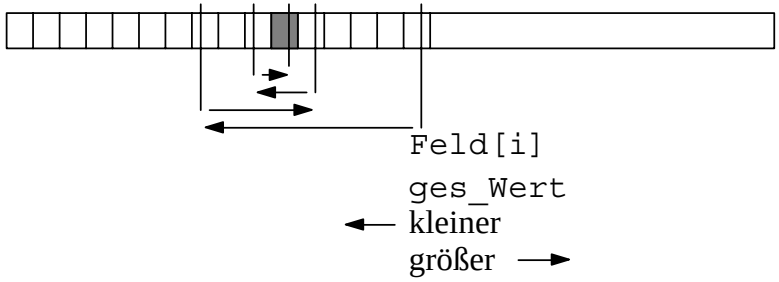
Im schlechtesten Fall muß das ganze Feld durchsucht werden: `linSuche` ist O(n)-Algorithmus (Anzahl der Vergleiche)

Im Mittel, wenn gesuchter Wert im Feld enthalten ist, benötigt man $\frac{n}{2}$ Vergleiche

Binäre Suche

– `linSuche` setzt keine Sortierung voraus
Sortierung bringt im Mittel keinen Effizienzgewinn

– Idee binäre Suche



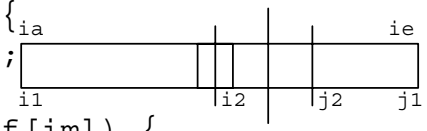
fortgesetzte Intervallschachtelung des Suchbereichs

– rekursives Programm (vereinfacht!)

```
void binSucheR(it_FT &f, item ges_Wert,
               index ia, index ie,
               index &ind, zust &z) {
 // Suche in Feldabschnitt f[ia]..f[ie],
 // 0<=ia, ie<=n-1
   index im; // mittlerer Index
   if (ia > ie) { // fuer Beendigung
      z = nicht_gef; return;
   } else {
      im = (ia + ie)/2;
   }
   if (f[im] == ges_Wert) {
      z = gef; ind = im;
   } else if (ges_Wert > f[im]) {
      binSucheR(f,ges_Wert,im+1,ie,ind,z);
   } else { // ges_Wert < f[im]
      binSucheR(f,ges_Wert,ia,im-1,ind,z);
   }
}
...
binSucheR(a, Wert, 0, n-1, i, status);
```

– iterative Fassung

```
void binSucheI(it_FT &f, item ges_Wert,
               index ia, index ie,
               index &ind, zust &z) {
   // Voraussetzung ia <= ie
   index i=ia, j=ie, im;
   while (true) {
      im = (i+j)/2; // mittlerer Index
      if (ges_Wert==f[im]){
         ind = im; z = gef;
         return; }
      else if (ges_Wert > f[im]) {
         i = im+1;} // Intervallschachtelung
      else { // ges_Wert < f[im]
         j = im-1;   // durch Indexveraend.
      }
      if (i>j) { // fuer Beendigung
         z = nicht_gef; return;
      }
   }
}
... // wie oben
binSucheI(a, Wert, 0, n-1, i, status);
```



– Vergleich: iterative Fassung einiges schneller!

- Messung: 1000 Elemente
  etwa 50% schneller als rekursive

– Analyse: Effizienzbetrachtung

In jedem Schritt wird die Hälfte ausgesondert:
Anzahl der noch zu betrachtenden Elemente
z.B. n=100, 49, 24, 11, 5, 2, 1 max. 7 Schritte
Anzahl zu untersuchender Fälle

gegenüber 100 im schlechtesten Fall
bzw. 50 im Mittel

Allgemein: $\lceil ld(n) \rceil$ Schritte im schlechtesten Fall

Wird durch Messung bestätigt:
1000 Elemente, alle Elemente 10000mal gesucht (10^7 Läufe):

`linSuche`		`binSuche`(iterativ)
1050 sec	Faktor 30	35 sec

Sortieren: Allgemeines

Präzisierung der Aufgabenstellung

– Problem:

In einer Datei befindet sich eine unbekannte Anzahl von Zahlen. Diese sollen gelesen und nach ihrer Größe sortiert ausgedruckt werden.

Unklarheiten:

- Mehrfachvorkommnisse: erlaubt, mehrfach aufgeführt?
- Sortierung: aufsteigend, absteigend?
- Sortierung in einem Feld: max. Anzahl bekannt? (internes Sortieren/externes Sortieren)
- Welcher Typ d. einzelnen Elemente: `int`, `float`, etc. ?
- Format: Wie ausdrucken?
- Feste Randbedingungen: Zahlen < größte in Basismaschine darstellbare Zahl

Vollständigkeit:

- Problemspezifikation unvollständig: Probleme bei Nutzung
- unvollst. Problemspezifikation
  Information aus anderer Quelle besorgen
  Spezifikation ergänzen

Widerspruchsfreiheit

– neue Formulierung:

In einer Datei befinden sich max. 100 nicht notwendigerweise verschiedene `unsigned`–Zahlen. Diese Zahlen sollen gelesen, der Größe nach aufsteigend sortiert und entsprechend ihrer Vorkommenshäufigkeit einzeln ausgegeben werde und zwar je 10 pro Zeile.

Programmierung durch schrittweise Verfeinerung

```
unsigned const dimVal=100;
typedef unsigned int item; // Aenderbarkeit
typedef int index;
typedef item it_FT[dimVal];
int n; it_FT a; unsigned pro_Z;

void main() {
   zahlenEinlesen   // Pseudo-
   zahlenSortieren  // anwei-
   zahlenDrucken    // sungen
}

{ // Zahlen einlesen:
   item x; n = 0;
   cin >> x;
   while(!cin.eof()) {
   // Zahl fuer Zahl in den Behaelter:
      a[n] = x; n++;
      if(n==dimVal) { break; }
      cin >> x;
   }
}

{ // Zahlen drucken:
   cout << "Die Zahlen in aufsteigender "
        << "Reihenfolge:" << endl;
   for (index k=0; k<n; k++) {
      cout << a[k] << " ";
      if (((k+1)%10)==0) { cout << endl; }
   }
}
```

Sortieren: Übersicht

– Hauptteil Sortieren
  einfache Verfahren: n^2
  gute Verfahren: n log n

  Sortierstrategien (s.u.):
    Einfügen
    Auswählen
    Vertauschen
    Verschmelzen

```
// Sortieren:
...

// Ergänzung der Deklarationen
// je nach Sortierverfahren (s.u.)

// Ausformulierung des Sortierens
// Da aufwendiger als andere Teile: Prozedur
```


– Def.: *Sortieren*:
  Anordnen einer Menge von Objekten in bestimmte Ordnung

  Gegeben $a_1, a_2, \ldots, a_n$
  Gesucht Anordnung $a_{k_1}, a_{k_2}, \ldots, a_{k_n}$ mit
    $f(a_{k_1}) \leq f(a_{k_2}) \leq \ldots \leq f(a_{k_n})$
  Schlüssel wird nicht erst berechnet, ist Komponente:

```
struct Info_Element {
    item key;
    .
    . // weitere Komponenten
    .
};
```

  `item` irgendein Typ mit Vergleichsoperationen, z.B. skalarer Typ, nicht notwendigerweise `unsigned int`;

– Sortieren also nicht Einteilung in Klassen (Sorten)
– Anwendungen:
- Voraussetzung für effizientes Suchen (s.o.)
- Datenbanksysteme, Betriebssysteme, Compiler, Anwendungen
- Ersatz für direkte Adressierung auf seq. Dateien

⇒ Verfahren mit geringem Speicher- und Laufzeitaufwand gesucht

– klassisches Kapitel (wie auch Suchen; nur Eingangsbeispiel)

für Leistungsanalyse von Algorithmen:
Tradeoff, Algorithmenstrukturen

– kann sein, daß $a_i = a_{i+1}$ etc.

Sortierverfahren heißen stabil, falls Reihenfolge von Elementen mit gleichem Schlüssel beim Sortieren unverändert bleibt.

Wichtig, da gleiche Elemente nach anderem Kriterium geordnet sein können.

– Klassifikation

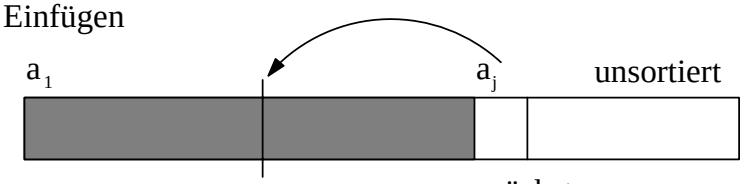
- internes Sortieren (auf Feldern) / externes Sortieren (auf seq. Dateien)
  (1) (2)
- direkte (einfache) Verfahren / verbesserte Verfahren
  (3) (4)
- stabile / instabile Verfahren
  (5) (6)
- einmaliges Sortieren(fester Datenbestand)
  (7)

  dauerndes Sortieren (veränderlicher Datenbestand)
  (8)

im folgenden (1), bedingt (3), (4), (5), (6), (7)
(8) kommt später im Zusammenhang mit Baumsuchen,
ext. Sortieren (2) wird nicht besprochen;

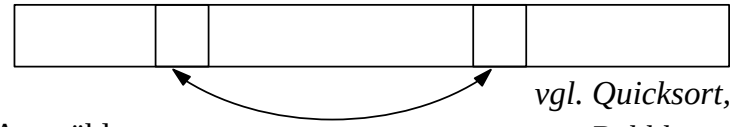
– Sortierprinzipien:

Einfügen



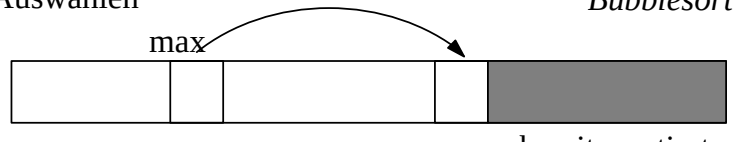
vgl. StraightInsertion

Austauschen



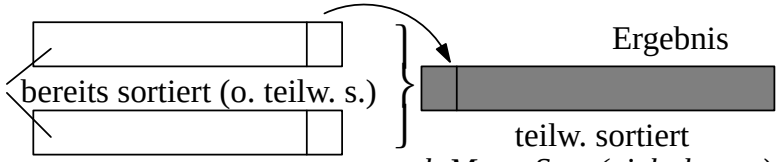
vgl. Quicksort, Bubblesort

Auswählen



vgl. StraightSelection

Mischen

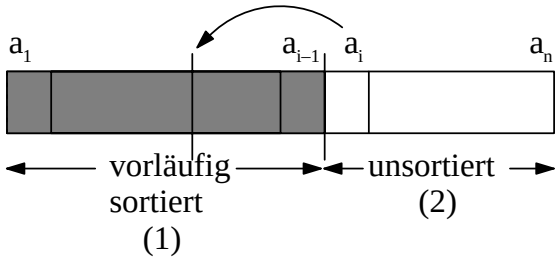


vgl. MergeSort (nicht bespr.)

Direkte Sortierverfahren

Sortieren durch Einfügen

– Idee noch einmal:



– Bsp.:

Anfangswerte	44	55	12	42	94	18	06	67
i=2	44	55	12	42	94	18	06	67
i=3	12	44	55	42	94	18	06	67
i=4	12	42	44	55	94	18	06	67
i=5	12	42	44	55	94	18	06	67
i=6	12	18	42	44	55	94	06	67
i=7	06	12	18	42	44	55	94	67
i=8	06	12	18	42	44	55	67	94

– Rohform:

```
for(int i=2; i<=n; i++) {
   // vom 2. Element an
   x=a[i];
   füge x geeignet in a[1]..a[i] ein
};
```


– nicht Stelle auffinden und dann verschieben,
  sondern Aufsuchen und verschieben in einem:
  von rechts nach links im Feldteil (1)
    Rechtsschieben, falls linkes Element noch größer;
    in die Lücke füllen, falls linkes Element kleiner / gleich
– Terminationsbedingungen
  • kleineres / gleiches Element gefunden
  • hinteres Ende erreicht

```
void StraightInsertion(it_FT &f, index ia,
                       index ie) {
 // direktes Verfahren, Sortieren durch
 // Einfuegen
   index j;
   item x;
   for (index i=ia+1; i<=ie; i++) {
      x = f[i]; // Zwischenspeicher
      j = i;
      // x einsortieren
      while ((j>ia) && (x<f[j-1])) {
         f[j] = f[j-1];
         j--;
      };
      f[j] = x;
   };
};
...
StraightInsertion(a, 0, n-1);
```

– Analyse: (C Anzahl der Vergleiche, M Anzahl der Bewegungen)

- Durchschnittliche Anzahl der Schlüsselvergleiche
  beim Einfügen von a_i ist $\frac{i}{2}$ (Mitte zwischen 1 und i–1)

$$\Rightarrow C_{\emptyset} = \sum_{i=2}^{n} C_i = \frac{1}{2}\left[\sum_{i=2}^{n} i\right] = \frac{1}{2}\left[\left[\sum_{i=1}^{n} i\right] - 1\right] = \frac{1}{2}\left[\frac{n(n+1)}{2} - 1\right] = \frac{n^2+n-2}{4}$$

- Durchschnittliche Anzahl von Bewegungen
  $M_i = C_i + 2$ (inkl. Marke setzen)

$$\Rightarrow M_{\emptyset} = \frac{n^2+n-2}{4} + (n-2+1)\cdot 2 = \frac{n^2+n+8n-2-8}{4} = \frac{n^2+9n-10}{4}$$

- bester Fall: alles geordnet

$$C_{best} = \sum_{i=2}^{n} 1 = n-1$$

$$M_{best} = \sum_{i=2}^{n} 3 = 3\cdot(n-1)$$

- schlechtester Fall: umgekehrt geordnet

$$C_{worst} = \sum_{i=2}^{n} i = \frac{n^2+n-2}{2}$$

$$M_{worst} = \frac{n^2+n-2}{2} + (n-2+1)\cdot 2 = \frac{n^2+5n-6}{2}$$

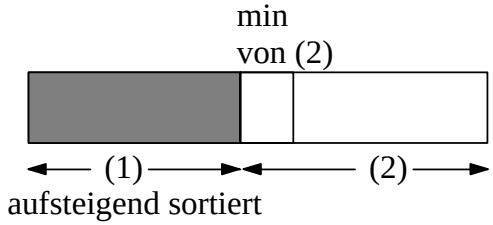
– Resumee:

typisches $O(n^2)$ - Verfahren, mit natürlichem Verhalten
(bester, durchschnittlicher, schlechtester Fall)

zur O-Notation, s.u.

Sortieren durch Auswählen

– direktes Auswählen




```
void StraightSelection(it_FT &f, index ia,
                       index ie) {
 // direktes Verfahren,
 // Sortieren durch Auswaehlen
   index k;
   item x;
   for (index i=ia; i<=ie-1; i++) {
      k = i; x = f[i];
      for (index j=i+1; j<=ie; j++) {
         if (f[j]<x) {
            k = j; x = f[j];
         }
      }
      f[k] = f[i]; f[i] = x;
   }
}
...
StraightSelection(a, 0, n-1);
```

– Analyse:

$C_{\substack{best \\ \emptyset \\ worst}} \in O(n^2)$ $\qquad$ $M_{best} = 3\cdot(n-1)$

$M_{worst} \in O(n^2)$

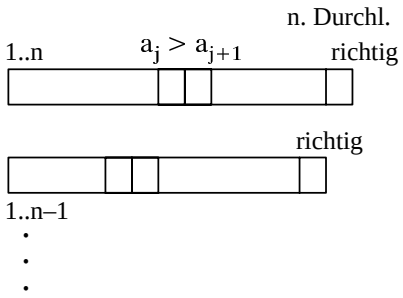
Somit: Wie kann die Anzahl der Vergleiche reduziert werden?
Kann von jedem Durchlauf mehr Information zurückbehalten werden, als das kleinste Element?
(siehe Heapsort in Literatur)

Verbesserung, Dokumentation, Test

Sortieren durch Vertauschen:

– Bubblesort

größere Blase steigt auf, bis sie durch eine noch größere Blase aufgehalten wird




```
void BubbleSort (it_FT &f, index ia,
                 index ie) {
// direktes Verfahren; Sortieren durch
// Vertauschen
bool getauscht;
item hilf;
   do {
      getauscht = false;
      for (index i=ia; i<ie; i++) {
         if (a[i]>a[i+1]) {
            // Vertausche
            hilf = a[i];
            a[i] = a[i+1];
            a[i+1] = hilf;
            getauscht = true;
         }
      }
   } while (getauscht);
}
```


vollständiges Programm

```
const dimVal=100;
typedef item it_FT[dimVal];
it_FT a;
index n=0;
item x;
unsigned pro_Z=10;

void Bubblesort(it_FT &f, index ia, index ie);

void main() { // Sortieren mit Bubblesort
   // Zahlen einlesen
   while (!cin.eof()) {
      cin >> x;
      a[n] = x; n++;
   };

   BubbleSort(a, 0, n-1);

   // Zahlen drucken:
   {  cout << "Die Zahlen in aufsteigender "
           << "Reihenfolge:" << endl;
      for (index j=0; j<=n-1; j++) {
         // Ausgabe 10 pro Zeile;
         cout << a[j] << " ";
         if (((j+1)%pro_Z)==0){cout << endl;};
      };
   };
}

// Hier Prozedur für BubbleSort
```

Verbesserungen

– Verbesserung 0. Schritt:
  eine noch unsinnigere Lösung wäre gewesen:
    n-maliger Durchlauf, unabhängig davon, ob im letzten Durchlauf vertauscht wurde
– Verbesserung 1. Schritt:

i	a					
undef.	16	33	94	82	6	58
1	16	33	94	82	6	58
2	16	33	94	82	6	58
3	16	33	82	94	6	58
4	16	33	82	6	94	58
5	16	33	82	6	58	94

1. Durchlauf größte Zahl am richtigen Platz
2. Durchlauf zweitgrößte Zahl am richtigen Platz usw.

- brauchen nur max. n–1 Durchläufe,
  letzte Zahl dann automatisch am richtigen Platz
  ⇒ letzter Durchlauf überflüssig (hatten wir schon)
- brauchen beim k-ten Durchlauf nur bis n–k zu laufen
- Abbruchkriterium *nicht* überflüssig: wollen rechtzeitig aufhören, wenn nicht mehr vertauscht wurde

– Verbesserung 2. Schritt:

k	i	a					
1	1	16	33	94	82	6	58
	...	...					
1	5	16	32	82	6	58	94
2	1	16	33	82	6	58	94
2	2	16	33	82	6	58	94
2	3	16	33	6	82	58	94
2	4	16	33	6	58	82	94
3	1	16	33	6	58	82	94
3	2	16	6	33	58	82	94
3	3	16	6	33	58	82	94

überflüssig bis n–k zu laufen, wenn im letzten Durchlauf nicht bis dorthin vertauscht wurde

– Neuer Sortierteil (Rumpf von Bubblesort)
  rechtes Ende pro Durchlauf nicht nur um 1 verkleinern:

```
...
int rechtesEnde, letztesI;
...
letztesI = ie;
do {
   rechtesEnde = letztesI; letztesI=ia-1;
   for (int i=0; i<rechtesEnde; i++) {
      if (f[i] > f[i+1]) {
         hilf = f[i]; f[i] = f[i+1];
         f[i+1] = hilf; letztesI = i;
      }
   }
} while (letztesI!=ia-1);
```

– Analyse: Effizienzbetrachtung Bubblesort

best–case–Analyse: geordnet
1 Schleifendurchlauf
n–1 Vergleiche, 0 Vertauschungen
hier Bubblesort gut!

worst–case–Analyse: umgekehrt geordnet
n–1 Durchläufe
k–ter Durchlauf:
innere Schleife (n–k)–mal vertauscht

$$\sum_{k=1}^{n-1}(n-k) = \sum_{k=1}^{n-1} k = \frac{n^2-n}{2}$$

Anzahl der Vergleiche/
Vertauschungen

$\Rightarrow$ Bubblesort $O(n^2)$

d.h. $\exists$ Konstante a, b, c mit

$$f^{WC}_{Zeit}(\text{Bubblesort, Feld\_der\_Laenge } n) < an^2 + bn + c$$

Konstanten nicht ausgerechnet: Vorsicht!

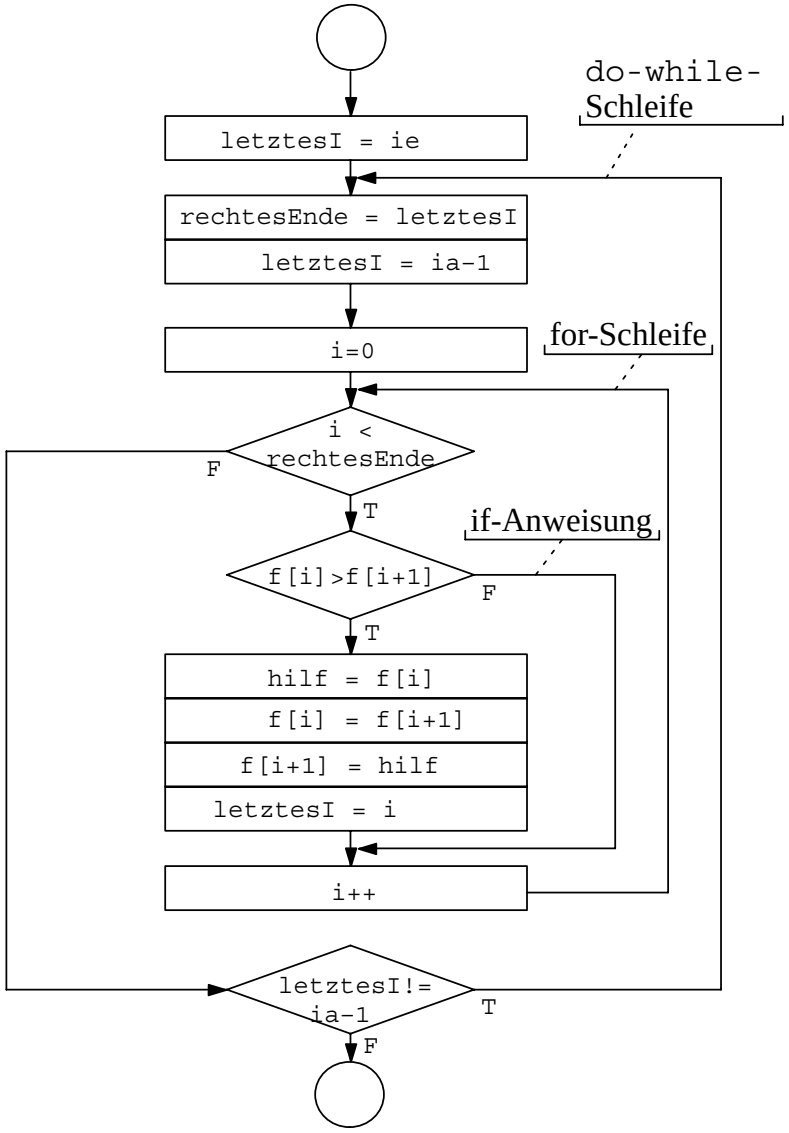
Es gibt bessere Verfahren: $O(n \log n)$

Dokumentation

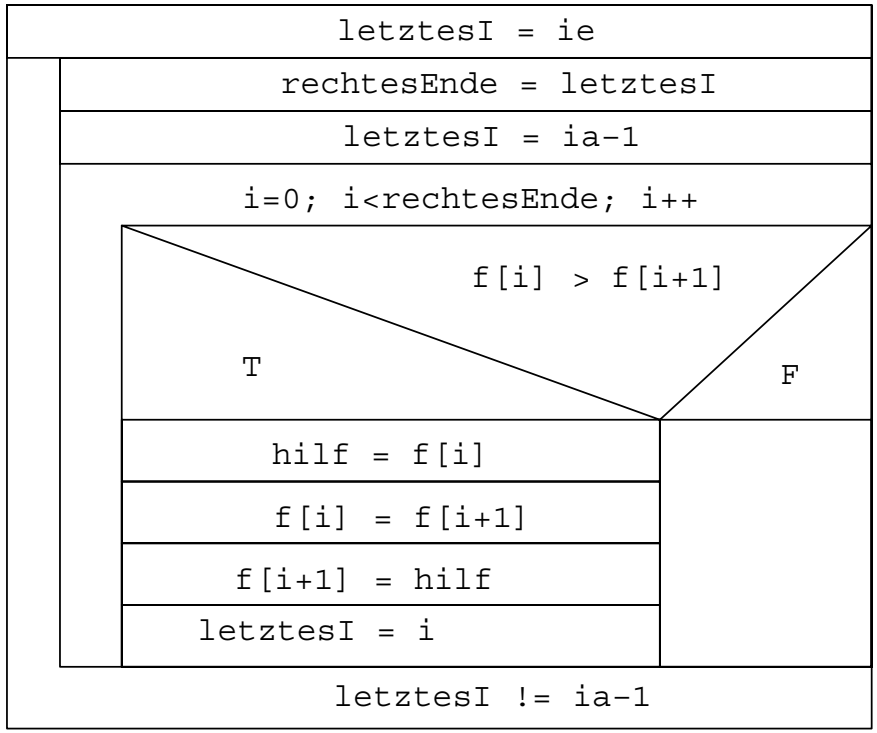
Allgemeines

- bisher: im Programm durch Kommentare
  Programm selbst durch Lesbarkeit
- "parallele" Notation: Flußdiagramme
  Struktogramme } (s.u.)
- große Projekte: Anforderungsdefinition
  Entwurfsspezifikation
  Rationales
  etc.

- Flußdiagramm zu Bubblesort-Rumpf



- Struktogramm zu Bubblesort-Rumpf



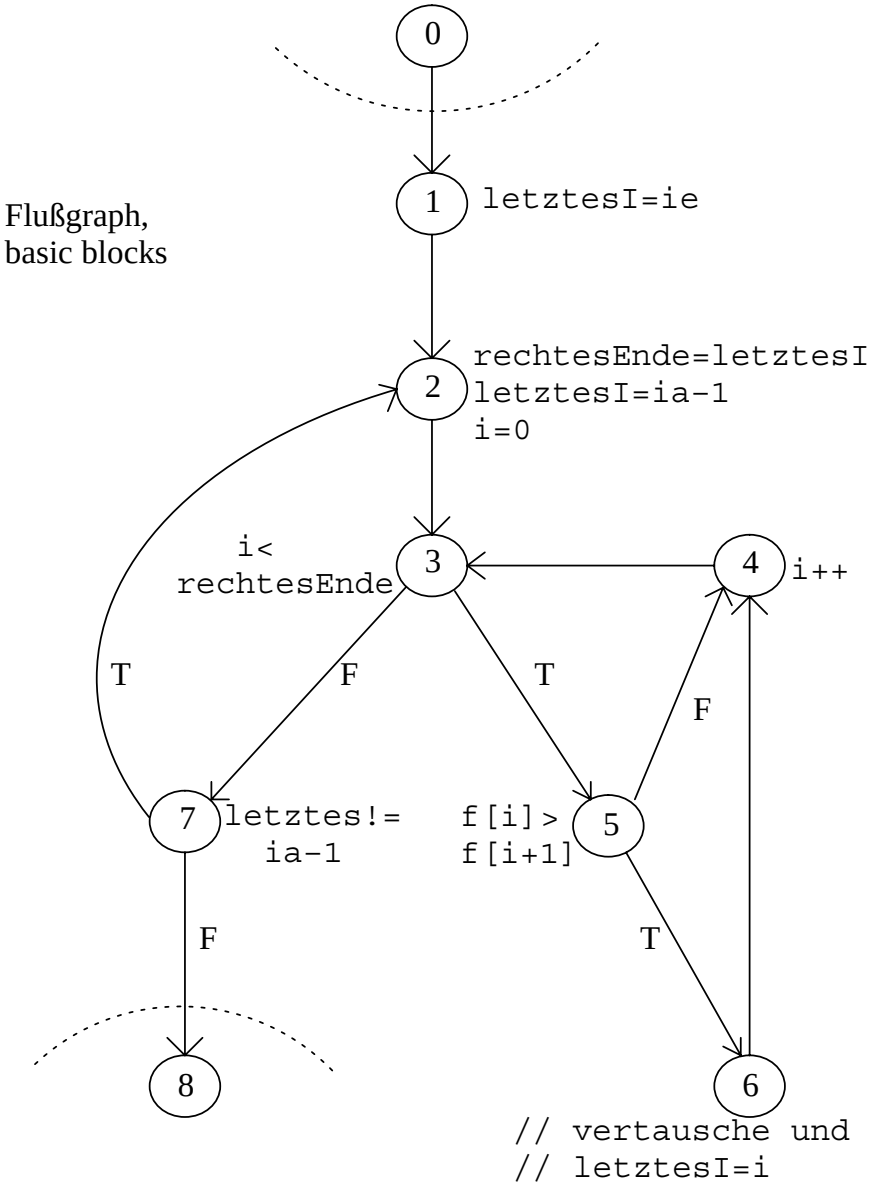
– Testen und Testdatenermittlung

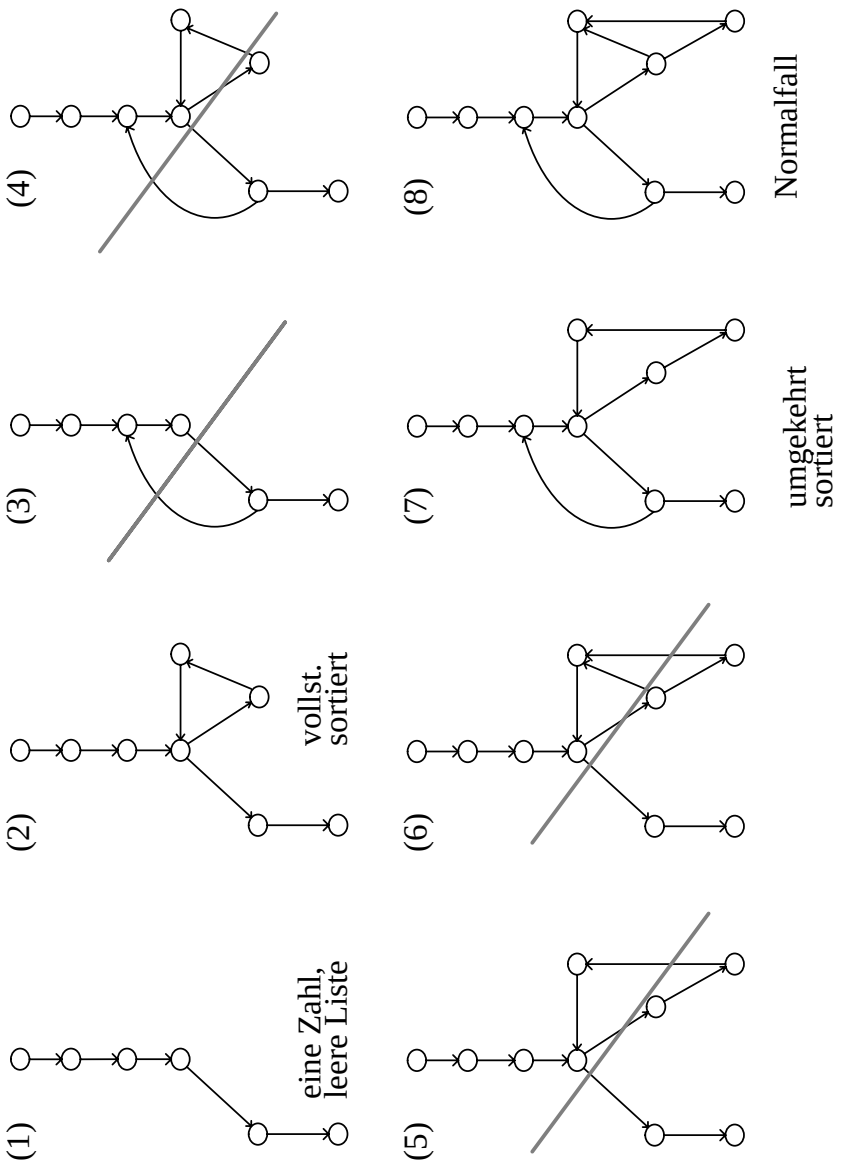
- black–box–Test (intuitiver Ansatz)
  z.B. Normalfall-Extremfall-Betrachtung

Normalfall	16	31	94	82	6	58
	...					
Extremfall	1	2	3	4	5	6
	6	5	4	3	2	1
	5	5	5	5	5	5
	6					
	leere Eingabe					

- white–box–Test

Flußgraph,
basic blocks





- Testen und Durchführbarkeit (white–box):

  Ermittlung aller dyn. Programmpfade nicht möglich!

  Ermittlung aller Programmpfade (bis auf Vielfache von Schleifendurchläufen, wie oben), nur bei kleinen Programmen möglich.

  Kantenüberdeckung und Bedingungsüberdeckung üblich (Überdeckungsverfahren nicht vorgeführt)!

- Testen und Korrektheit:

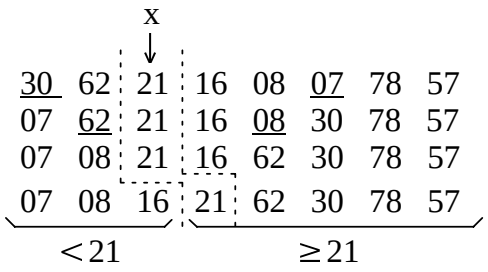
  “Durch Testen kann nur die Anwesenheit aber nicht die Abwesenheit von Fehlern gezeigt werden.” (Dijkstra)

Effizientes Verfahren: Quicksort

Entwicklungsidee

– Können wir ein verbessertes Verfahren finden, das die Anzahl der Vertauschungen verkleinert?
Ansatz: Bewegungen eines falsch plazierten Elements um größere Strecke, nicht nur eine Stelle nach links oder rechts

– Idee:
Quicksort (Hoare 1962): gebräuchlichste interne Sortiermethode
Zeiger i,j (anfangs i=ia (=0), j=ie (=n–1)); wähle Index *eines* Elementes x aus.
Durchlaufe von links bis f_i mit $f_i > x$, von rechts bis $f_j < x$;
Vertausche beide Elemente f_i, f_j; setze i=i+1, j=j–1;
Wiederhole bis i>j.

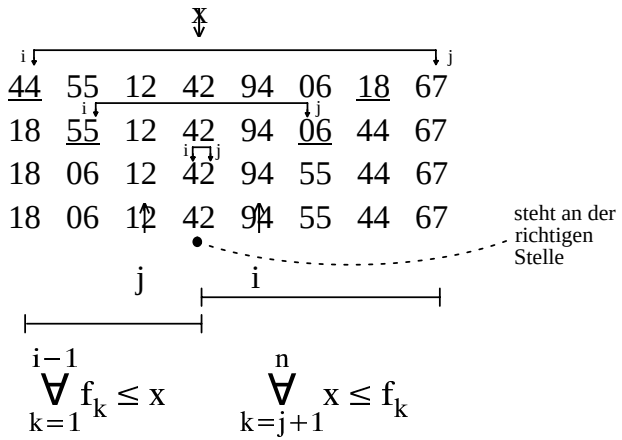


Anwendung auf beide entstandenen Teilintervalle etc.

Index nach Durchlauf muß nicht mit x übereinstimmen!

– erste Fassung, Rumpf der Prozedur:

```
item x, hilf;
index ia=0, ie=n-1;

// Partition: zerteilt das
// zu sortierende Feld in zwei Abschnitte
{  index i=ia, j=ie;
   waehle ein zufaelliges Element x aus
   f[0]..f[n-1] aus;
   while(i<=j) {
      while (f[i]<x) { i++; }
      while (x<f[j]) { j--; }
      if (i<=j) {
         hilf=f[i]; f[i]=f[j]; f[j]=hilf;
         i++; j--;
      }
   }
}
```



Invariante der while-Anweisung

anfangs klar

bei jedem Vertauschungsschritt/Indexweiterschieben erhalten

=> Ende: Zerlegung in zwei Intervalle mit Schlüsseln kleiner-gleich bzw. Schlüsseln größer-gleich

Lösung: rekursive Anwendung auf Teilintervalle

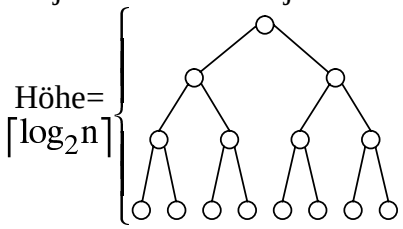
– Programm

```
...
void QuickSort(it_FT &f, index ia, index ie);
  // Prozedurspezifikation
void Sort(it_FT &f, index L, index R);
  // Prozedurspezifikation:
  // Anschluß irgendein Suchverfahren

void main() {
   ...
   // Einlesen: ...
   Sort(a, 0, n-1);
   /// Ausgabe: ...
}
void Sort (it_FT &f, index L, index R) {
   QuickSort(f, L, R);
}
void QuickSort(it_FT &f,index ia,index ie) {
   index i=ia, j=ie; item x, hilf;
   {  x=f[(ia+ie)/2];//Element mit mittl. Index
      while (i<=j) {
         while (f[i]<x) { i++; }
         while (x<f[j]) { j--; }
         if (i<=j) {
            hilf=f[i]; f[i]=f[j]; f[j]=hilf;
            i++; j--;
         }
      }
      if (ia<j) { QuickSort(f, ia, j); }
      if (i<ie) { QuickSort(f, i, ie); }
   }
}
```

Analyse

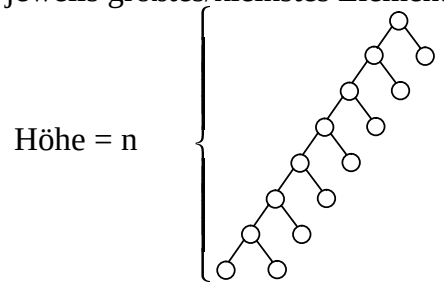
– Auswahl von x entscheidend

– best case:
x jeweils "Mittelobjekt"



auf jedem Niveau n Vergleiche
$\Rightarrow \; C_{best} \in O(n \log_2 n)$

– worst case:
x jeweils größtes/kleinstes Element



$\Rightarrow \; C_{worst} \in O(n^{2})$

– average case Laufzeit: $O(n \log_2 n)$ Sehr gut!
allerdings sollte n nicht zu klein sein
($\rightarrow$ dir. Methode)

– Auswahl des mittleren Elements
- Laufzeit gut bis auf pathologische Fälle
- Mittelobjekt (Median)
  ≤ Hälfte der Schlüssel
  ≥ Hälfte der Schlüssel
- exakte Bestimmung würde Sortieren voraussetzen. Wie können wir ungefähr bestimmen und damit schlechtesten Fall verbessern?
- Näherung in obiger Lösung
  Wähle Mittelobjekt aus $\left\{a_l, a_{\lfloor \frac{l+r}{2} \rfloor}, a_r\right\}$,
  d.h. entsprechenden Index

4 Module, Datenabstraktion (Programmstrukturierung ohne Objektorientierung)

Lernziele:

Module als Bausteine der Architektur
Datenabstraktionsprinzip
Datenobjektmodule, Datentypmodule (Klassen)
Simulation von Modulen in C++

Module und Architektur

Module als Bausteine

– bisher Hauptprogramm und Prozeduren

– Prozedur: ein Dienst an der Schnittstelle
  Module: mehrere Dienste, die “zusammengehören”

– große Programme: Aufteilung in Module (und Teilsysteme)
  Gründe: Entwurf vor Realisierung
  Überprüfung gegen Problemstellung vor Programmerstellung
  Überprüfung Programmerstellung gegen Entwurfsspezifikation
  Wartbarkeit (Anpaßbarkeit, Portierbarkeit)
  Arbeitsteilung
  Qualitätssicherung
  Dokumentation
  etc.

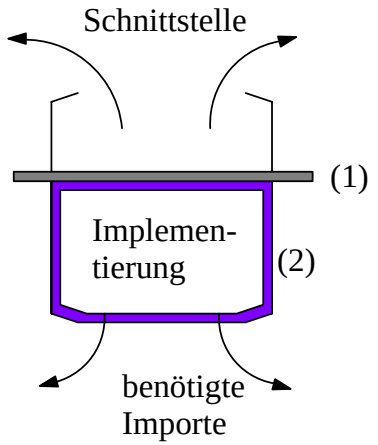
– Export: Was der Modul für andere zur Verfügung stellt
  Import: Was der Modul zu seiner Realisierung an Hilfe braucht
  bisher z. B. Import von `iostream.h`

– Module eröffnen
  - Erstellung von Programmbausteinen und Ablage in Programmbibliothek
  - getrennte Bearbeitung und Übersetzung

Module und Abstraktion

– Modul ist Abstraktion (1)
  von Implementierungsdetails des Moduls
  von hierzu benötigten Importen

  Bsp.: `iostream.h`,
  wie bisher verwendet



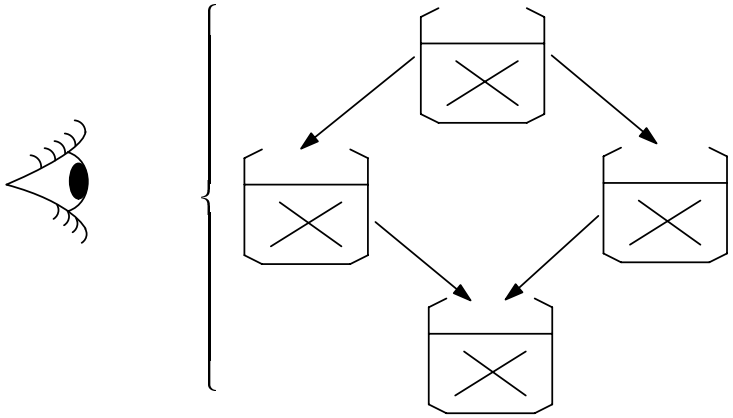
– Modul ist Kapsel (Information Hiding) (2)
  verbirgt Interna
  Programmiersprache verhindert ggfs. (nicht C++)
  Zugriff auf diese: Sie sind nicht sichtbar.

– Module
  Definitionsteil (Schnittstelle)
  –> Export, öffentlich zugänglich
  Implementationsteil (Rumpf des Moduls)
  –> verborgen
  beide Teile sind getrennt übersetzbar
  benötigen weitere Dienste (Importe)

Module in der Architektur

– Modul ist Teil einer Softwarearchitektur
  (→ Vorlesung Softwaretechnik, insb. Architektur-modellierung)
  diese erhält Struktur durch
    Verwendung bestimmter Arten von Modulen
    Verwendung bestimmter Arten von Importen
    Konsistenzbedingungen
    ist Hierarchie bestehend aus vielen Bausteinen

– besondere Art der Abstraktion



alle Bausteine aufgeführt,
insb. auch in tiefen Schichten,
Details der Bausteine (Rümpfe) nicht betrachtet
idealisierte Vorstellung;
geht praktisch nicht anders

Schnittstelle und Rumpf

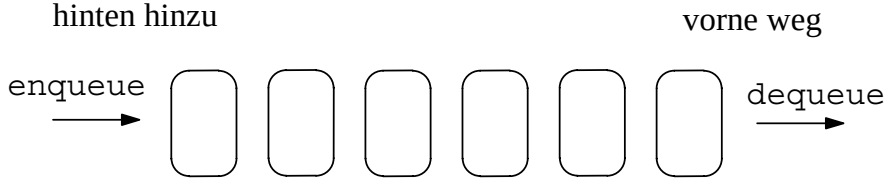
Schnittstellen von Modulen

– Schnittstelle enthält exportierte Objekte
  (Ressourcen, die Modul anderen zur Verfügung stellt)
  nämlich:
    Konstante (Vorsicht → Datenabstraktion)
    Variable (Vorsicht → Datenabstraktion)
    Typen (Vorsicht → Datenabstraktion)
    Prozeduren (nur Schnittstellen, Rümpfe im Implementationsteil)

– Gültigkeit/Sichtbarkeit in allen importierten Modulen
  Konstante
  Variable mit Feinstruktur } Vorsicht:
  Typen mit Feinstruktur } Veränderung
    (bei transparenten (offenen) Typen)
  Prozeduren (Schnittstelle, aber nicht Rumpf)

– Typen
  transparente (s.o.)
  opake: Beschränkung auf Zeiger
    Details im Rumpf gekapselt

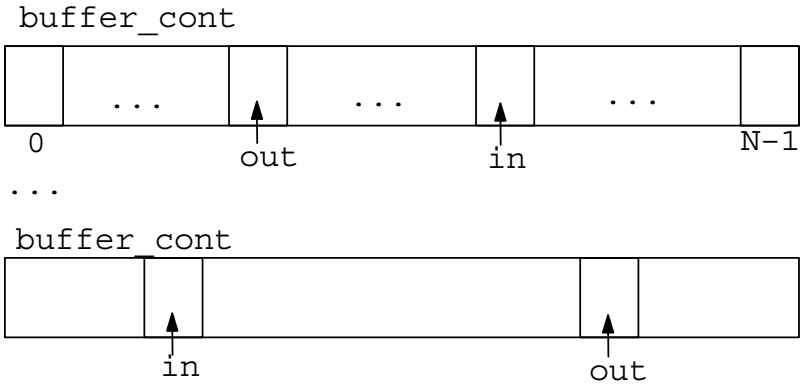
– Schnittstelle sollte logisches Protokoll definieren,
  z.B. Puffer (buffer, queue)

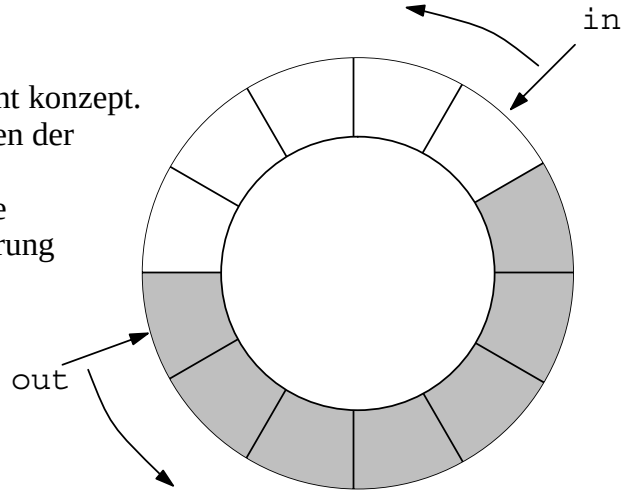


Zugriffsstrategie FIFO

Rumpf eines Moduls

– Rumpf (Implementation) enthält Realisierung der Schnittstelle
  - Realisierung z.B. zirkuläre Speicherung





entspricht konzept. Verkleben der Enden: zirkuläre Speicherung

Realisierung in C++

```
// Schnittstelle (interface, protocol)=====

void enqueue(item x);
void dequeue(item &x);
bool nonEmpty();
bool nonFull();
// Sei item ein Datentyp fuer die Elemente
// des Puffers (austauschbar).
// enqueue fuegt hinten ein Element dazu
// dequeue nimmt vorne ein Element weg und
// liefert dabei dessen Wert.
// Voraussetzung von enqueue:
// Datenstruktur hat noch Platz;
// fuer dequeue: Datenstruktur nicht leer.
// ---------------------------------------
```

```
// Rumpf (body, implementation) ------------
// Datenstruktur mit zirkulaerer Speicherung
// Realisierung der Schnittstelle ist darauf
// abgestimmt

unsigned int const N=100; //Behaeltergroesse
static int In=0;  // Werte: 0,...,N-1;
static int Out=0; // Verwaltung der Enden
static int n=0;   // Wert: 0,..,N; Anzahl
                  // der Elemente
typedef itemT it_BT[N];
static it_BT bc;  // Behaelter fuer Buffer
static bool nempty=false, nfull=true;

void enqueue (itemT x) {
   if (n<N) {
      bc[In]=x; In=(In+1)%N;
      n++; nfull=(n<N); nempty=true;
   }
}

void dequeue (itemT &x) {
   if (n>0) {
      x=bc[Out]; Out=(Out+1)%N;
      n--; nempty=(n>0); nfull=true;
   }
}

bool nonEmpty() {
   return nempty;
}

bool nonFull() {
   return nfull;
}
// ========================================
```


– andere Realisierung (z.B. verkettete Liste)
  ohne Auswirkung auf alle importierenden Module

– Information Hiding
  Realisierung:
    1) andere interne Datenstrukturen
    2) andere Implementation der Operationen
    3) in der Regel andere importierte Module

– Vorbedingung für Benutzung durch Klienten
  `nonEmpty` für `dequeue`
  `nonFull` für `enqueue`

Datenabstraktionsprinzip, Datenobjektmodule

Datenabstraktionsprinzip

– Datenstruktur mit Zugriffsoperation eine Einheit: Modul

– Zugriffsoperation nach außen:
  Funktionale (logische) Sicht der Schnittstelle:
    Zugriffsoperationen (Prototypen)

– Informationsverbergung auf “logischer” Ebene
  Realisierung der Datenstruktur } verborgen
  Realisierung der Zugriffsoperation } verborgen
    benötigte Importe } vgl. (1) von Abb. 1

– Datenstruktur mit dieser Eigenschaft:
    abstraktes Datenobjekt
  Modul, der eine solche Datenstruktur realisiert:
    abstraktes Datenobjektmodul

– Bei Datenabstraktion sind stets Datenstrukturen mit Gedächtnis im Spiel

– keine Auswirkung auf Klienten bei Änderung der Realisierungstechnik

– Bsp. `buffer`
  Schnittstelle (unverändert)

  Realisierungsentscheidung
    Datenstrukturen
    Importe
    $\Rightarrow$ Realisierungstechniken für Zugriffsoperationen

– Datenabstraktion insbesondere bei Verwendung von Zeigern: Gefahr lokalisieren

– Datenabstraktion macht Schnittstelle zwischen Moduln “dünn”
  Überlegung:
    globaler Speicher für Puffer
    Pufferoperationen direkt realisiert an allen Stellen der Verwendung

– jede komplexe “Datenstruktur” ist in der Systemarchitektur zu sehen: jeweils Modul dafür

– Realisierungsbandbreite:
  auf Laufzeitkeller bei lokalem Modul
  auf statischem Speicher bei Bibliothekseinheit
  auf der Halde bei Verwendung von Zeigern
  auf Sekundärspeicher
  auf Knoten eines Netzes von Rechnern

– Schnittstellengestaltung von DA–Bausteinen
  Sicherheitsfragen (s.o.)
  return–Parameter (Zustandsparameter für Berechnung)

Datenobjektmodule

– bisher: `buffer` ist abstraktes Datenobjekt (Datenkapsel)
  abstrakt: über sauberes Protokoll mit Zugriffsoperationen gehandhabt

– abstr. Datenobjekt hat Gedächtnis (Zustand)
  wird bei Zugriffsoperationen `enqueue`, `dequeue` verändert;
  `nonFull` und `nonEmpty` fragen Zustand ab

– Verwendung (Datenobjekt über Namen der Op. identifiziert)

```
if (nonFull()) {
   enqueue(it);
} else { Error(...);
}
```

} Verwendung des abstr. Protokolls; Realisierung nicht zu sehen

– abstr. Datenobjekt ist Baustein der Architektur

– Verwendung für Grunddatenstrukturen, die nur einmal vorhanden sind:
  Datenbestände/Datentöpfe in sauberer Form

Hilfsmittel zur getrennten Übersetzung

Dateien für Schnittstellen und Rümpfe

– Unabhängige Übersetzung:
  Nicht das ganze Programm, sondern Teile hiervon werden einzeln geprüft.
  Konsistenz zwischen den Teilen (unzureichend) durch Binder

– Somit unabhängige Übersetzung:
  Separate Übersetzung ohne (mit teilweiser) Querprüfung
  Gewünscht: getrennte Übersetzung:
    separate Übersetzung mit Querprüfung
  separat übersetzte Einheiten = Dateien in C++

– separate Übersetzung wegen getrennter Bearbeitung und zur Verminderung der Recompilationszeit

– Header-Files
  Schnittstellenbeschreibungen von Funktionen, Modulen (zur Simulation von Modulschnittstellen)
  (Dateiname z.B. `a.h`)

– Implementation-Files
  enthalten Rümpfe bei Modulen, Funktionsdeklarationen bei Funktionen
  (zur Simulation von Modul–Rümpfen)
  (Dateiname z.B. `a.cpp`)

– Main-Files
  für das Hauptprogramm (Dateiname z.B. `mainprog.cpp`)

– Bibliotheks-Files
  enthalten Zusammenfassung vordefinierter Bausteine
  (zur Simulation von Teilsystemen)
  (Dateiname z.B. `stdlib.h`)

Steueranweisungen für das Programmiersystem

1) `#include`

Verwendung aber Neuübersetzung

```
#include ”a.cpp”
#include ”b.cpp”
void main() {
   proc_a1();  // Funktionsaufruf
   proc_a2();
   return func_b();
}
```


Einbinden vorübersetzter Teile

```
// a.h
void proc_a1();
void proc_a2();
```

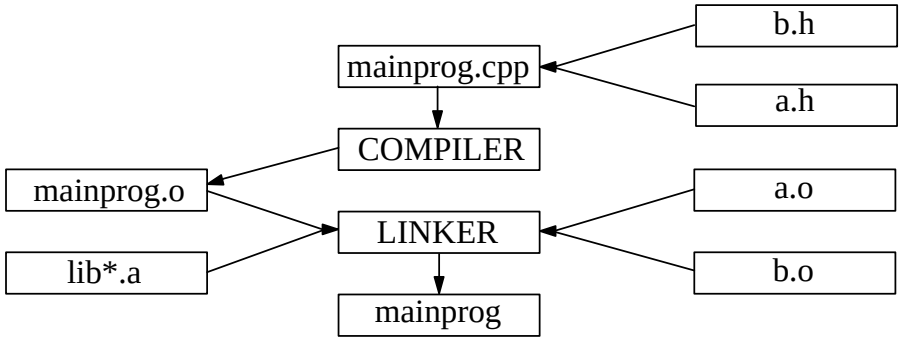
```
// a.cpp
#include ”a.h”
void proc_a1() {
   // Programmcode zu proc_a1
}
void proc_a2() {
   // Programmcode zu proc_a2
}
```

```
// b.h
int func_b();
```

```
// b.cpp
#include “b.h”
int func_b() {
   // Programmcode zu func_b
}
```

```
// mainprog.cpp
#include ”a.h”
#include ”b.h”
void main() {
   proc_a1();
   proc_a2();
   return func_b();
}
```

Steuerung der Übersetzung



Durch “Makefiles” oder “Projekte” werden größere Softwarevorhaben handhabbar.

2) `#ifndef, #define`

`#include "a.h"` kann zu Problemen führen, wenn die in `a.h` enthaltene Schnittstelle an mehreren Stellen benötigt und eingefügt wird.

```
// c.h
#ifndef __INC_C_H__
#define __INC_C_H__

void proc_c();
int const C_MAX = 100;

#endif
```

Nutzen dies später bei allen Schnittstellenangaben, um für Importe geeignetes Schema zu finden.

Dateiübergreifende Gültigkeit und Sichtbarkeit

`extern` vor Deklaration: ist in anderer Datei festgelegt.
`static` vor Deklaration: Gültigkeit auf aktuelle Datei beschränkt.

Alle anderen Deklarationen (`auto`) in Prozeduren/Funktionen:
Variablendeklarationen auf Laufzeitkeller angelegt.

Funktionale Module

Allgemeines

– aktivitätsorientierte Module
EA Verhalten
kein Gedächtnis

– für Transformation (z.B. Compiler)
für komplexe Berechnung (z.B. numerische Integration)

– Anzahl der Dienste:
ein Dienst: Prozedur als Spezialfall
mehrere Dienste: funkt. Modul, Funktionsmodul

– Welche Dienste zusammenfassen:
gleiches Parametertypprofil
evtl. zusätzlich ähnliche Realisierung

Beispiel in C++

– Schnittstelle

```
// function module Trigonometry ===========

#ifndef __INC_TRIGONOMETRY_H__
#define __INC_TRIGONOMETRY_H__

float acos(float x); // Arcus Cosinus
float asin(float x); // Arcus Sinus
float atan(float x); // Arcus Tangens
float cos(float x);  // Cosinus
float cosh(float x); // Cosinus Hyperbolicus
float sin(float x);  // Sinus
float sinh(float x); // Sinus Hyperbolicus
float tan(float X);  // Tangens
float tanh(float X); // Tangens Hyperbolicus
#endif
// end interface Trigonometry--------------
```

– Rumpf

```
// body Trigonometry-----------------------

#include "trigonometry.h" // Uebereinstimmung
                          // des Rumpfes mit
float acos(float x) {     // der Schnittstel-
   ...                    // le muss der Pro-
}                         // grammierer
float asin(float x) {     // gewaehrleisten;
   ...
}
  .
  .
  .
// end body Trigonometry====================
```


Handhabung von Importen

– Importe zur Kennzeichnung der Verwendung eines Teils der Schnittstelle eines anderen Moduls

– Bsp. Modul `Grafik`
in dessen Rumpf wird von `Trigonometry` `sin` und `cos` verwendet

```
// with Trigonometry import sin, cos;
#include "trigonometry.h"

// body Grafik-----------------------------
// Der Programmierer ist verantwortlich
// dafuer, dass nur sin und cos benutzt
// werden.

void rotate(float x, float y, float w,
             float &newx, float &newy) {
   // Berechnet die Koordinaten des Punktes
   // (x,y) nach einer Rotation um den
   // Ursprung um den Winkel w (Bogenmass)
   // in (newx, newy)

   newx = x*cos(w) - y*sin(w);
   newy = x*sin(w) + y*cos(w);
}

...

// end body Grafik==========================
```

Datentypmodule als Klassen

Motivation

– jetzige Realisierung: Modul ist Gedächtnis (Datenobjektmodul)
  Was tun, wenn mehrere Datenobjekte benötigt werden?
  Quelltextvervielfältigung?

– Aussehen normaler Datenobjektdeklaration:

```
typedef ElementT BT[N];
BT b_obj;          //BT fuer Buffer Type
```

über normale Typdeklaration definiert

Struktur von `BT` und somit `b_obj` ist auf bel. Strukturniveau bekannt

gegen Datenabstraktionsidee!

– Wollen stattdessen:

```
...
if (nonfull(b_obj)) {  // wird spaeter in
   enqueue(b_obj, it1) // C++ anders notiert
   ...
} else {
   ...
}
```

– Brauchen Modul der
  "Typbezeichner"
  Zugriffsoperationen für Objekte dieses Typs } exportiert
  und sonst nichts: abstr. Datentypmodul

– kein Modul für Gedächtnis
  sondern für Schablone für Gedächtnisse

– Operationen haben einen "Parameter mehr",
  für das entspr. abstrakte Datenobjekt

– Erzeugung der abstrakten Datenobjekte
  über Deklarationen
  über Erzeugungsoperationen
    "analog" zu `new` bei Haldenverwaltung

– Methodikregel:
    abstrakte Datenobjekte werden ausschließlich über Zugriffsoperationen verändert

Realisierung des Puffers als ADT (Klasse)

– Schnittstelle

```
// ADT fuer Puffer ========================
#ifndef __INC_BUFFER_TYPE_H__
#define __INC_BUFFER_TYPE_H__

class Buffer_Type { // Interface
// Der Modulname ist auch der Typ, der
// ausserhalb verwendet wird; Objekte koennen
// damit deklariert und erzeugt werden.
// Handhabeung der Objekte nur ueber
// Zugriffsoperationen.
// Diese haben die folgende Semantik: ...

   // Protokoll fuer Verwendung ausserhalb
   // durch Klienten
public:
   Buffer_Type();
   void enqueue(item x);
   void dequeue(item &x);
   bool nonempty();
   bool nonfull();

private: // physische Schnittstelle f. Compiler
   static unsigned const N=10;
   int In;              // Dieser Teil ist
   int Out;             // aussen nicht zu
   unsigned n;          // verwenden.
   itemT bc[N];          // Er beschreibt den
   bool nempty;         // Aufbau der Objekte.
   bool nfull;
};
#endif Buffer_Type_h
// end interface Buffer_Type------------------
```


– Rumpf der Klasse

```
// body ADT Buffer_Type-------------------

#include "Buffer_Type.h"

Buffer_Type::Buffer_Type() {
   // Erzeugung und Initialisierung
   // der Objekte
   In=0; Out=0; n=0;
   nempty=false; nfull=true;
}

void Buffer_Type::enqueue(itemT x) {
   if (n<N) {
      bc[In]=x; In=(In+1)%N; n++;
      nfull=(n<N); nempty=true;
   }
}

void Buffer_Type::dequeue(itemT &x) {
   if (n>0) {
      x=bc[Out]; Out=(Out+1)%N; n--;
      nempty=(n>0); nfull=true;
   }
}

bool Buffer_Type::nonempty() {
   return nempty;
}

bool Buffer_Type::nonfull () {
   return nfull;
}

// end body Buffer_Type====================
```

Verwendung der Klasse in anderen Bausteinen

```
//---Erzeuger---------------------
#include "Buffer_Type.h"

void erzeuge(Buffer_Type buffer) {
  ...
  if(buffer.nonfull()) {
    buffer.enqueue(daten);
  }
}

//---Verbraucher------------------
#include "Buffer_Type.h" // nonempty,dequeue

...


//---Kontrollmodul----------------
#include "Buffer_Type.h"  // Konstruktor
#include "Erzeuger.h"     // erzeuge
#include "Verbraucher.h"  // verbrauche

int main()
{
  Buffer_Type buffer;

  erzeuge(buffer);
  verbrauche(buffer);

  ...

  return 0;
}
```


Module und Simulation in C++

Simulation von Modulen

– funktionale Module
  abstr. Datenobjektmodule } Schnittstellen waren zu simulieren

  abstr. Datentypmodule: Schnittstellen durch Klassenkonstrukt zusammengefaßt

– Rümpfe aller dieser Modularten
  existieren nur als gedankliche Zusammenfassungen;
  in jeweils einer Implementationsdatei

– Schnittstellen → Header–Files
  Rümpfe → Implementation–Files
  damit Module simulierbar, wie oben gezeigt

– Importe simuliert durch `#include "M.h"`

– zur Vermeidung von Problemen beim mehrfachen Import

```
#ifndef __INC_M_H__
#define __INC_M_H__
...
#endif
```

Disziplin bei Simulation

– Durch `#include` wird die ganze Schnittstelle importiert;
  daß nur die angegebenen verwendet werden: Disziplin

– Übereinstimmung zwischen Schnittstellen und Rumpf:
  Disziplin
    Schnittstelle muß realisiert werden,
    dabei dürfen weitere lok. Hilfsmittel deklariert werden

– Abschottung des Rumpfes: Disziplin
  Datenstrukturen bei ADOs dürfen außen nicht
    verwendet werden,
    wird durch `static` garantiert
  bei ADTs (Klassen) wird Zugriff durch `private`
    verhindert

5 Datenstrukturen: Listen, Bäume, Graphen

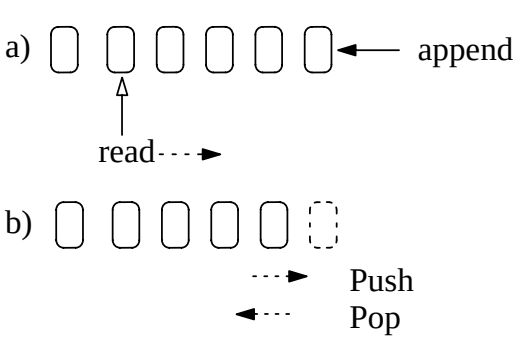
Lernziele:
  Kennenlernen von Standard-Datenstrukturen
  Datenstrukturen als Anwendung von Datenabstraktion
    Realisierung hängt von Schnittstelle ab
    Realisierungsvarianten für eine Schnittstelle
  Algorithmen, die Datenabstraktionsbausteine verwenden
  verschiedene Tradeoffs, z.B. Laufzeit und Speicherplatz

Lineare Listen

Lineare Listen mit verschiedenen Zugriffsmöglichkeiten

– im folgenden ist Elementtyp ein Text, in C vom Typ `char*`

– wir betrachten im folgenden Abschnitt 2 Listenarten



Liste hinten beschrieben von vorne gelesen (Schlange, FiFo Datenstruktur)

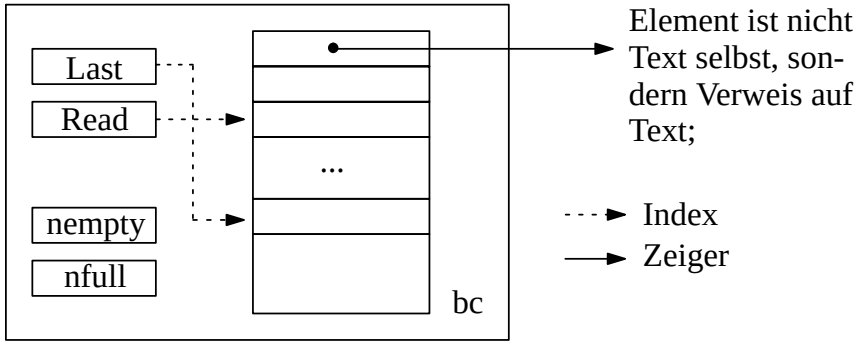
Keller (stack): LiFo Datenstruktur

– Schnittstelle der Append–Liste als ADT

```
// Append_Liste A_List_Type als ADT ========
#ifndef __INC_A_LIST_TYPE__
#define __INC_A_LIST_TYPE__

class A_List_Type {   // Interface
// append verlaengert die Liste hinten;
// reset setzt einen Lesezeiger auf das
// vordere Ende;
// read liest ein Element und setzt den
// Lesezeiger auf das naechste Element;

public:
   A_List_Type();        // Konstruktor
   void append(char *x);// x wird kopiert
   void reset();         // zuruecksetzen
   void read(char *&x); // lesen, weiters.
   bool nonempty();      // fuer read anfangs
   bool nonfull();       // fuer append
   bool is_end();        // fuer read

private:
   unsigned int const N=999;
                        //Behaeltergroesse
   int Last;   // Index letztes Element
   int Read;   // Index aktuelles Element
   char *bc[N];// Behaelter als Feld
   bool nempty;
   bool nfull;
};

#endif
// end A_List_Type Interface ---------------
```

Sequentielle Realisierung von A_List_Type

– Realisierungsskizze:




```
// A_List_Type body ----------------------
A_List_Type::A_List_Type() {
   Last = 0; Read = 0;
   nempty = false; nfull = true;
}

void A_List_Type::append(char *x) {
   if (Last < N) {
      bc[Last] = x; Last++;
      nfull = (Last<N); nempty = true;
   }
}
...

void A_List_Type::read(char *&x) {
   if (Read != Last) {
      x = bc[Read]; Read++;
   }
}
...
// end A_List_Type body ====================
```


Diskussion

– `append`, `read`: O(1)

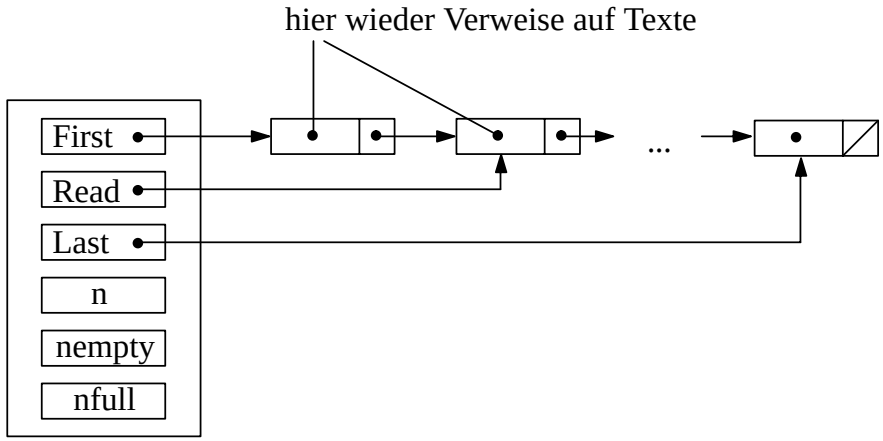
– sequentielle Realisierung geeignet
  wenn nur `append(el)` benötigt wird
  wenn die Gesamtzahl der Elemente zur Entwicklungszeit bekannt

– weitere mögliche Operationen von `A_List_Type` (später nicht benötigt)
  `read(p)`: lies Element an Position `p`:
    sequ. Realisierung mit Feld erweiterbar: O(1)
  `insert(el, p)`: Füge `el` an Position `p` ein:
    Verschieben des entspr. Teilfelds nach unten: O(n)
  `delete(p)`: Löschen des Elements an Position `p`:
    Verscheiben des entspr. Teilfeldes nach oben: O(n)
    insb. `delete(1)`: Verschieben aller Elemente
      `deletefirst`
  (vergleiche jedoch zirkuläre Speicherung von oben)

– können `insert(el,p)`, `delete(p)` bzw. `deletefirst()` verbessert werden?

Verkettete (verzeigerte) Realisierung linearer Listen

– Realisierungsskizze



– Änderung des privaten Teils der Schnittstelle
Hinzunahme von `deletefirst()`

```
public:
   B_List_Type(); // public-Teil genau
   ...            // wie A_List_Type
   void deletefirst();
   ...
private:
  struct B_List_item { // Struktur der
    B_List_item *next; // Listenelemente
    char        *item;
  };
  B_List_item *First;  // Struktur des
  B_List_item *Last;   // Verwaltungsblocks
  B_List_item *Read;
  bool nempty;
  bool nfull;
};
```


– Änderung des Rumpfes der Liste

```
#include "blist.h"

B_List_Type::B_List_Type() {
   First = NULL; Last = NULL; Read = NULL;
   nempty = false; nfull = true;
}

void B_List_Type::deletefirst() {
   if (First != NULL) {
      B_List_item hilf = First;
      First = First->next;
      delete hilf;
   }
}

void B_List_Type::append(char *x) {
   // Speicher holen fuer neues Element
   B_List_item *newitem = new B_List_item;

   // neues Element initialisieren
   newitem->next = NULL;
   newitem->item = x;

   // und an die Liste anhaengen
   if (! nempty) { // Liste noch leer
      First = newitem;
      nempty = true;
   } else {
      Last->next = newitem;
   }
   Last = newitem;
};
```

```
void B_List_Type::reset() {
   Read = First;
}


void B_List_Type::read(char *&x) {
   if (Read != NULL) {
      x = Read->item;
      Read = Read->next;
   }
}


bool B_List_Type::nonempty() {
   return nempty;
}


bool B_List_Type::nonfull() {
   return nfull;
}

bool B_List_Type::is_end() {
   return (Read == NULL);
}


// end body B_List ---------------------
```


Diskussion Zeigerrealisierung

– Aufwand `deletefirst`: O(1)
Preis: Speicherplatzaufwand für Zeigerfelder
Tradeoff: Laufzeit–Speicherplatz

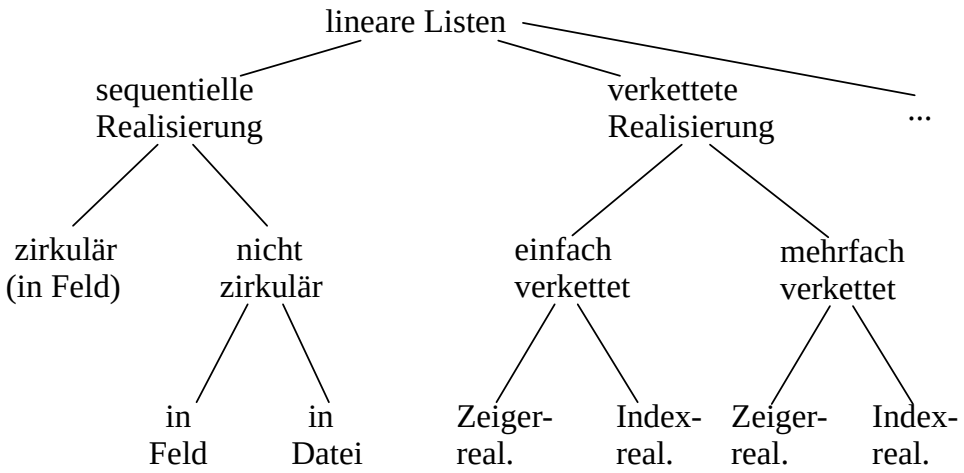
– Aufwand `delete(p)`: O(n)
wegen Bewegung an die richtige Stelle

– wurde mit `read` bereits diese Navigation gemacht,
dann Aufwand O(1)

– Wahl der Realisierung hängt von gewünschten Zugriffsoperationen ab:
Generalthema dieses Teils der Vorlesung

– Bei Löschung eines Elements muß man vor dem Element stehen:
um diese Besonderheit zu vermeiden und nun auch zurücklaufen zu können (z.B. `read_prev`) geht man besser zu doppelt verketteten Listen über (Übungsaufgabe)

Übersicht: Realisierungstechniken für lineare Listen



weitere Realisierungsarten für Listen geordnet nach Schlüsseln: Assoziativspeicher, Binärbaum, etc. } z.T. später

Verwendung linearer Listen

Benötigte Listenarten

– Brauchen lineare Listen mit Anfügen hinten und Lesen von vorne (haben hierfür sequ. und verkettete Realisierung kennengelernt)

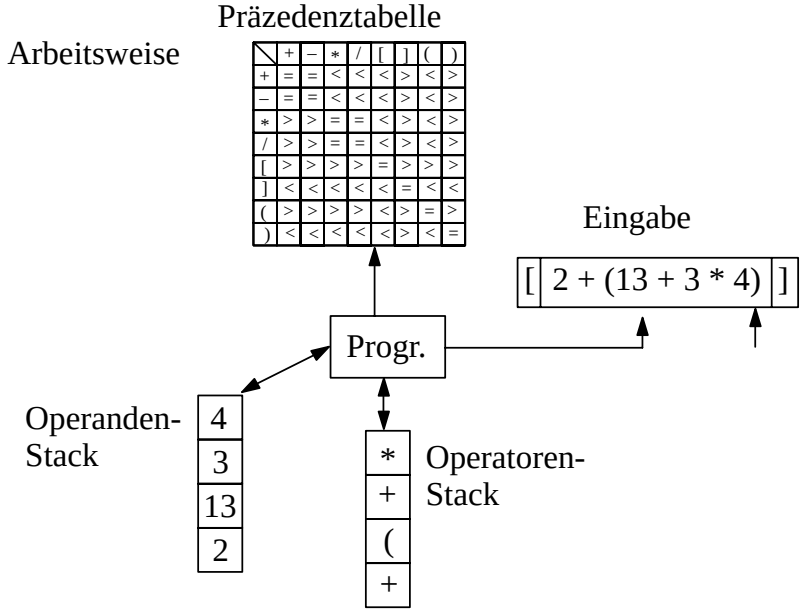
– Brauchen lin. Listen als LiFo-Datenstruktur (Keller, Stapel, Stack) mit Texten als Element

```
// ADT Stack_Type Interface ================
#ifndef __INC_STACK_H__
#define __INC_STACK_H__

class Stack_Type {
// ...
public:
   Stack_Type();
   void push(int x);
   void pop();
   void read_top(int &x);
   bool isempty();
   bool isfull();

private:

};
#endif
// end Stack_Type Interface ----------------

// body Stack_Type ------------------------
// wieder sequ. oder verkettet(Uebung)
...
// end body Stack_Type =====================
```

Aufgabe: Auswertung einfacher ganzzahliger Ausdrücke

– Ausdrücke in Infixnotation enthalten:

Literale:	13, 40
binäre Operatoren:	+, –, *, /
Begrenzer:	(,)



↘	+	–	*	/	[	]	(	)
+	=	=	<	<	<	>	<	>
–	=	=	<	<	<	>	<	>
*	>	>	=	=	<	>	<	>
/	>	>	=	=	<	>	<	>
[	>	>	>	>	=	>	>	>
]	<	<	<	<	<	=	<	<
(	>	>	>	>	<	>	=	>
)	<	<	<	<	<	>	<	=

– Tokens sind:

PL=–1; MI=–2; MU=–3; DI=–4; BO=–5;BC=–6;
Start=–7; END=–8
oder ganzzahlige Werte für Literale

Brauchen `Scan`, das Token als ganzz. Wert zurückliefert,
d.h. entweder Wert für Begrenzer/Operator
oder int-Wert für ganzzahliges Literal

Brauchen Pozedur `precendence`, die für zwei Operatoren/Begrenzer Operatorenvorrang bestimmt, also z.B.

+, *	$\lessdot$
+,+	$\doteq$
*,+	$\gtrdot$


```
#include <iostream>

#include "alist.h"
#include "stack.h"

enum OpDel {
  PL=-1, MI=-2, MU=-3, DI=-4, BO=-5, BC=-6,
  START=-7, END=-8
};

enum ComRes {GT, EQ, LT}; // fuer Praez.Vergl.

void Scanner(char *ausdruck, A_List *token);
int Auswerten(A_List *token);

int main(int argc, char *argv[]) {
  int wert;
  char *ausdruck = argv[1];
  A_List *token = new A_List();
  token->append(START);
  // virtueller Operator mit hoechster
  // Prioritaet als Anfangszeichen
  Scanner( ausdruck, token );
  // Erzeugt aus dem Ausdruck eine Liste
  // von Token
  token->append(END);
  // virtueller Operator mit niedrigster
  // Prioritaet als Endezeichen
  wert = Auswerten( token );
  cout << "Der Wert des Ausdrucks ";
  cout << ausdruck << " ist " << wert;
  cout << endl;
  delete token;
}
```

```
int Scan(char *token);
void Scanner(char *ausdruck, A_List *token){
...
};
ComRes precedence(int o1, int o2);

int Auswerten(A_List *token) {
  int wert, t;
  int lastop=START;

  Stack_Type *op = new Stack_Type();
  Stack_Type *lit = new Stack_Type();

  token->read(t);
  while (true) {
    if (t >= 0) {
    // Token t ist ein Literal
      lit->push(t);
      // und kommt auf den Literalstack
      token->read(t);
    } else { // sonst ist t ein Operator
      int curop = t;

      // Abbruchbedingung der Endlosschleife
      if (START==lastop && END==curop) {
        break;
      }

      if (LT==precedence(lastop, curop)) {
      // aktueller Operator hat groessere
      // Praezedenz und kommt auf den Stack
        op->push(curop); lastop = curop;
        token->read(t);
```

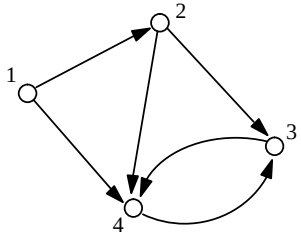


```
      } else {
      // der letzte Operator auf dem Stack
      // kann ausgewertet werden
        int l1, l2;

        switch( lastop ) {
        case START:
          op->push(curop); lastop=curop;
          token->read(t); break;
        case BO: // '('
          if (BC==curop) {
          // Klammern weglassen
            op->pop(); op->read_top(lastop);
            token->read(t);
          } else {
            op->push(curop); lastop=curop;
            token->read(t);
          }
          break;
        case PL: // '+'
          op->pop(); op->read_top(lastop);
          lit->read_top(l2); lit->pop();
          lit->read_top(l1); lit->pop();
         lit->push(l1+l2); break;
        // MI, MU, DI entsprechend
        ...
        }
      }
    }
  }
  // auf dem Literalstack liegt das Ergebnis
  lit->read_top(wert);
  delete op;
  delete lit;
  return wert;
}
```

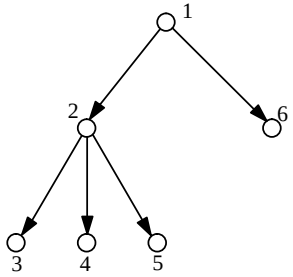
Bäume

Allgemeines

– Def.
gerichteter Graph G ist G = (K, E) mit
K endl. Menge (*Knoten*)
E ⊆ K x K (*Kanten*)
k_2 *Nachf.* von k_1
k_1 *Vorg.* von k_2
Folge von Knoten
Kz = (k_1, ..., k_{n+1}) heißt (gerichteter) *Kantenzug*
g.d.w. (k_i, k_{i+1})∈E
n *Länge* von Kz, Anzahl der Kanten
Zyklus g.d.w. Kantenzug geschlossen, d.h. $k_1=k_{n+1}$
(Auslauf) *Grad* von k ist Anzahl der Nachfolgerknoten
(weiteres später)



– Def. (*Wurzel*) *Baum* ist zyklenfreier Graph, wobei jeder Knoten höchstens einen Vorgänger hat und bei dem jeder Knoten von bestimmten Knoten (Wurzel) über Kantenzug erreichbar ist.
(Def. *Vater, Söhne, Brüder, Teilbäume, Blätter, innere Knoten, binärer Baum, ternärer Baum*)



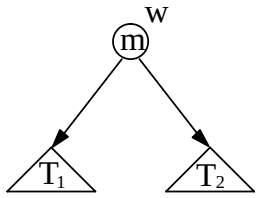
– Def. Sei k Knoten eines Baumes T.
Stufe (Niveau) von k def. durch

$$lev(k) = \begin{cases} 1, & k \text{ ist Wurzel} \\ lev(Vorg(k)) + 1, & \text{sonst} \end{cases}$$

Höhe $h(T) := \max_{k \text{ Knoten von } T} (lev(k))$

– Def. (Baumdurchläufe binärer Bäume):

- Baum leer bzw. besteht der Baum nur aus isolierten Knoten, dann ist die leere Liste bzw. die Liste aus dem einz. Knoten die Preorder-, Inorder- und Postorder-Liste



- Sonst ist
Preorder-Durchlauf Pr(T) def. durch Pr(T):=(m, $Pr(T_1)$,$Pr(T_2)$)
Inorder- In(T) In(T):=($In(T_1)$,m,$In(T_2)$)
Postorder- Po(T) Po(T):=($Po(T_1)$,$Po(T_2)$,m))

mit $m = f_{lab}(W)$

– Bem.: Syntaxbaum Inorder-Durchlauf ⇒ Infixnotation
Preoreder- ⇒ Präfixnotation
Postorder- ⇒ Postfixnotation

ADT Baum und Verwendung

ADT Baum: Schnittstelle

– Operationen eines ADT Tree
(bel. geordn. mark. Baum):
`Parent(n,T)` liefert Vater von n in T, für Wurzel undef.
`leftmostChild(n,T)` liefert ersten Nachfolger von n in T; undef. für Blatt
`rightSibling(n,T)` liefert nächsten Bruder; undef. für letzten Sohn
`Label(n,T)` liefert Markierung von n
$\texttt{Create}_i\texttt{(m, } T_1\texttt{, } T_2\texttt{, ..., } T_i\texttt{)}$ liefert Baum mit Wurzel k, die mit m markiert ist, und T_1, T_2, ... als Nachfolger von links nach rechts besitzt. Spezialfall i=0 erzeugt isolierten Knoten.
`Root(T)` liefert Wurzelknoten, undef. für leeren Baum
`MakeNull(T)` macht T zum leeren Baum
`isEmpty(T)` liefert `true` für leeren Baum, sonst `false`

– Operationen eines ADT BinTree
`LeftTree(n,T), RightTree(n,T)`
...

Baumdurchlauf unter Verwendung des ADT Tree

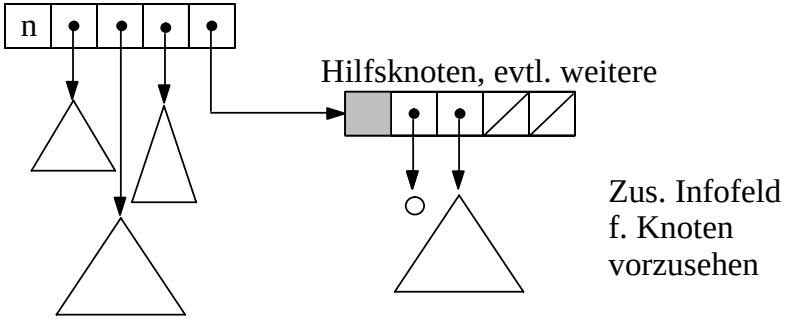
```
// nichtrekursive Fassung des
// Preorder-Durchlaufs
void NRPreorder(Node n, Tree t) {
  Node m;
  NodeStack s;
  // ADO mit Eintraegen des Typs Node,
  // enthaelt jeweils den Pfad von der
  // Wurzel zu aktuellem Knoten
  if (!isempty(t)) {
    m = n;
    while (true) {
      if (m!=NULL) {
        Ausgabe(Label(m, t));
        push(m, s);
        // gehe zum naechsten Nachfolger
        // ueber
        m = leftmostChild(m, t); }
      else {
        // Abarbeitung des Pfades auf dem
        // Keller ist jetzt fertig
        if (isEmpty(s)) { return; };

        // fahre mit Geschwisterknoten des
        // obersten Kellerelements fort:
        m = rightSibling(read_top(s), t);
        pop(s);
      }
    }
  }
}
```

– noch effizientere nichtrekursive Fassung gewinnt man, wenn man den Keller selbst im Baum ablegt.
  Hierzu muß
  - der ADT Tree auch die Op. Parent (Übergang zum Vater) zur Verfügung stellen (war oben nicht nötig),
  - Verwaltungsinfo am Knoten abgelegt werden, wie z. B. “schon besucht”

Realisierungsvarianten für den ADT

Listenelemente für Knoten und Nachfolger

– Knoten und Verweise auf Nachfolger werden als Listenelemente abgelegt.
  Hat jeder Knoten festen Auslaufgrad: entsprechende Knotenelemente, sonst Überlaufelemente



– Realisierung mit C++ - Zeigern (oder Indizes), ersteres etwa:

```
struct Knoten;
typedef Knoten* ZK;
struct Knoten {
   unsigned Anzahl;
   item Info;
   ZK Zeiger[4];
};
```

– damit die Operationen leftmostChild, rightSibling, Label, $Create_i$ leicht zu realisieren
  Operation MakeNull: Freigeben aller Knotenelemente

– Speicherplatzeffizienz:
  Sei T ein Baum vom Grade k mit n Knoten, realisiert durch Knoten mit je k Zeigerfeldern

  $\Rightarrow n*k - (n-1) = n*(k-1) + 1$ sind frei

  Zeigerdef. Anz. Kanten

  damit

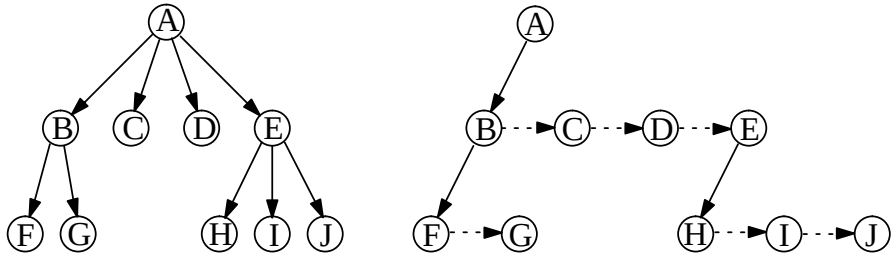
  ternäre Bäume: mehr als 2/3 der Zeiger NIL

  binäre Bäume: Hälfte NIL

– Bilanz noch einmal verbesserbar durch anderen Datentyp für Blatt (ohne Zeiger auf Nachfolger):
  Dann sind Erweiterungsoperationen etwas komplizierter.

Abspeicherung eines bel. Baumes durch einen binären

– Umwandlung interessant wegen obiger Speicherplatzüberlegung:

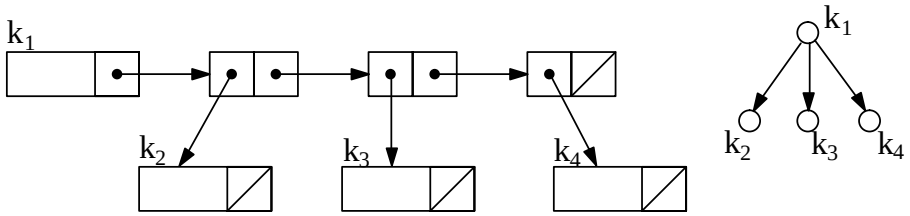


2+4+4+ 11 NIL-Felder
1+4+1+4+4+4=31 NIL-Felder
bei 4 Zeigerfeldern

sog. leftmostChild-rightSibling-Implementation

Knoten- und Kantenelemente

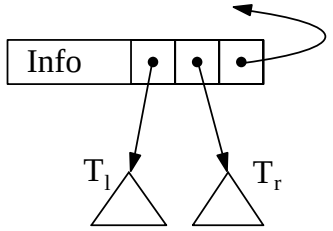
– andere Implementationstechnik: zusätzlich Kantenelemente



Speicherbilanz nicht günstiger als bei obiger Binärbaumrealisierung:
n+(n–1)*2 gegenüber 2n Zeigerfeldern

Verfädelung

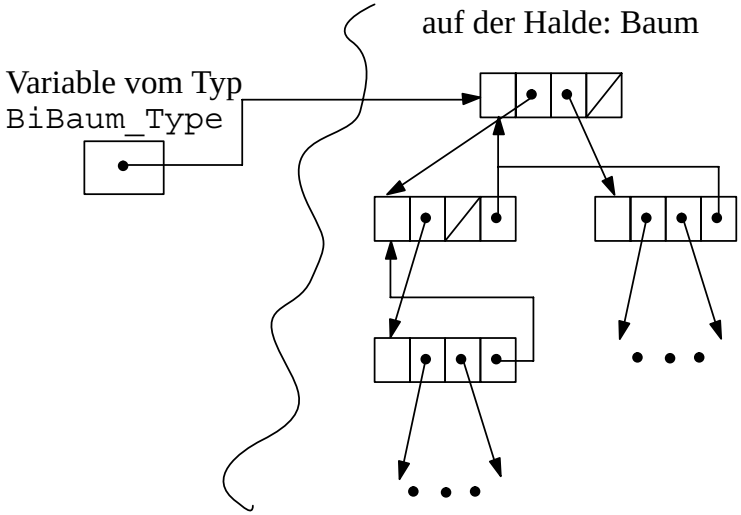
– alle bisher besprochenen Implementationen erlauben keine effiziente Implementation der Operation Parent
Lösung: Zusätzlicher Zeiger bei jedem Knoten auf den Vater.
Für Binärbaum mit Zeigerrealisierung etwa:



– Struktur der Knotenelemente:

```
struct Knoten;

typedef Knoten* ZK;

struct Knoten {
   item Info;
   ZK liBaum;
   ZK reBaum;
   ZK Vater;
};

typedef ZK BiBaum_Type;
```

Bei ADT als Klasse ist der Typ die Klasse selbst (Übung!)
Wie sieht die Lösung aus, wenn auch `Knoten` zu Klasse wird (Übung!)



– Resumee:

Leftmost_Child-Right_Sibling-Realisierung günstiger als Knotenelemente mit Überlaufelementen

besser als Knoten- und Kantenelemente

ggfs. mit Verfädelung

Realisierung auf Halde oder mit Indizes (Cursorrealisierung)

Verwendung binärer Bäume als Suchbäume

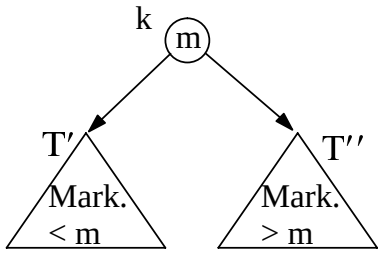
Binäre Suchbäume

– Def. binärer Suchbaum ist bin. mark. Baum T = (K,E) mit

$\forall k \in K$ gilt

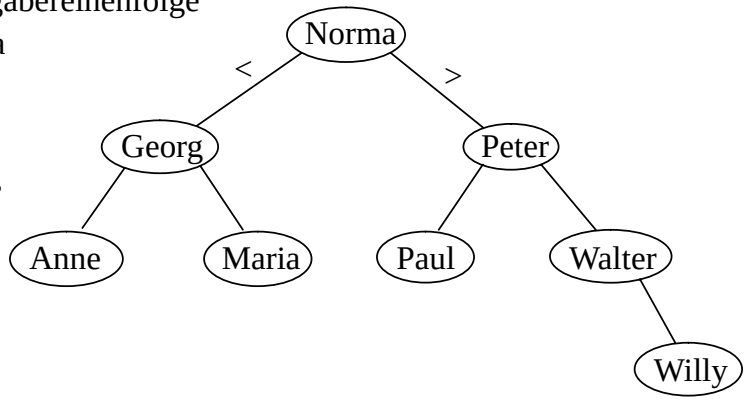
$$\begin{cases} f_l(k') < f_l(k) \;\; \forall k' \in K' \text{ mit } LeftTree(k,T) = T' = (K',E') \\ f_l(k'') > f_l(k) \;\; \forall k'' \in K'' \text{ mit } RightTree(k,T) = T'' = (K'',E'') \end{cases}$$

$f_l(k)$ liefere die Info von k zum Vergleich (label)



– Bsp. Eingabereihenfolge

Norma
Peter
Paul
Georg
Walter
Maria
Anne
Willy



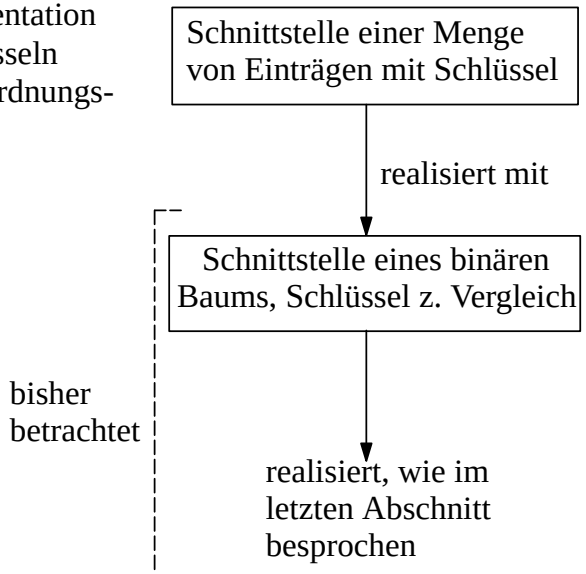
– Bem.

- < bez. sich auf ganzz. Schl., oder lexikograph. Ordnung auf Zeichenketten
- Inorder-Reihenfolge liefert totale Ordnung
- Baumaufbau (”Ausgeglichenheit”) hängt von der Eingabereihenfolge ab: z.B. liefert Eingabe in totaler Ordnung degenerierten Baum (Pfad)

Menge realisiert mit binärem Suchbaum

– jetzt “inkrementelles” Sortieren: Bei jeder Veränderung wird gleich der veränderte Teil neu sortiert,
Suche macht von der jeweils aktuellen Sortierung Gebrauch.
Anwendung: z.B. Symboltabelle

– jetzt Baum als Implementation von Mengen mit Schlüsseln auf denen eine totale Ordnungsrelation gilt



– Realisieren
alle folg. Algorithmen mithilfe eines ADTs `BinTree_Type`, der z.B. Op. `LabelNode()`, `Root()`, `MakeNull()`, `IsEmpty()`, `rightTree()`, `leftTree()`, `InsertRightNode(m)` (als Spezialfall von $\texttt{Create}_i\texttt{(...)}$ von oben) und einen Klassenbezeichner `BinTree_Type` exportiert

Vorteil: Sind bei Alg. zur Suchbaumbehandlung vollst. unabhängig von der Realisierungsart des Binärbaums (Zeigerreal., Indexreal., etc.)

– Suche in Menge `M`, die intern als binärer Suchbaum org. ist.
`Contains` ist Schnittstellenoperation eines entspr. Moduls (Klasse)

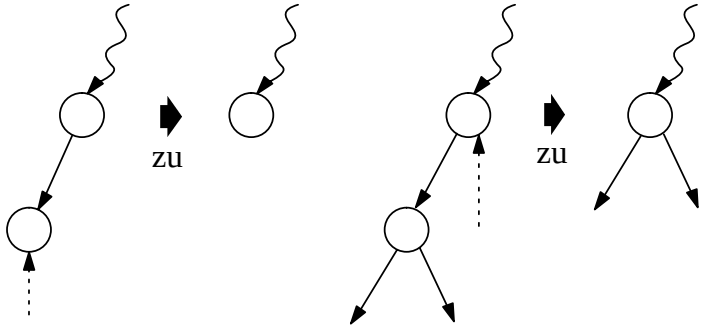
```
class A_Menge_Type { // interface
// ...

public:
   A_Menge_Type(); // Konstruktor
   bool Contains(elType x);
   ...
private:
   BiBaum_Type StTree; // Zeiger auf intern
                       // benutzten bin. Suchbaum
};
...

bool A_Menge_Type::Contains(elType x) {
   BiBaum_Type B=StTree;
   while (B!=NULL) {
      if (B->LabelNode() > x) {
         B=B->rightTree(); }
      else if (B->labelNode() < x) {
         B=B->leftTree(); }
      else {
         return true; // gefunden
      }
   }
   return false; // nicht gefunden; stehen
                 // auf Knoten unterhalb dem
                 // einzufuegen waere
}
```

– Analyse Suchen
  - für ausgegl. Suchbaum $\in \quad O(\log_2 n)$
    (n Anz. d. Einträge = Anz. d. Knoten des Suchbaums)
  - für degenerierten Suchbaum $\in O(n)$
  - beweisbar: deg. Baum selten. Bei zufäll. Reihenfolge:
    $C_{\emptyset} = 2 \ln n$
    (Was heißt zufällig bei alphanumerischen Schlüsseln?)
  - Komplexitätsaussage gilt auch f. Einfügen, Löschen und Minimumssuche

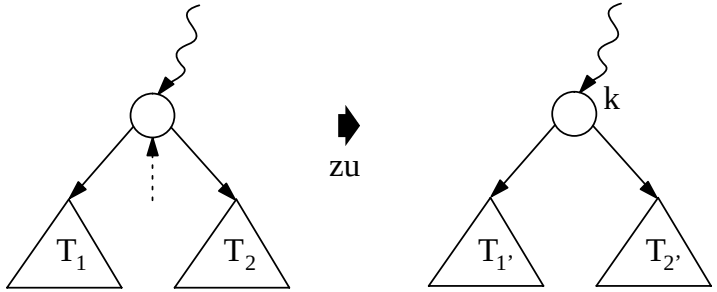
Löschen und Einfügen in bin. Suchbäumen

– im Rumpf von Menge als lok. Prozedur
  Binärbaum-Schnittstelle muß entspr. Hilfsmittel anbieten

– Löschen in binärem Suchbaum
  - Vorüberlegungen:
    a) einfach, falls Blatt oder innerer Knoten mit nur einem Nachfolger



    b) Knoten mit zwei Nachfolgern

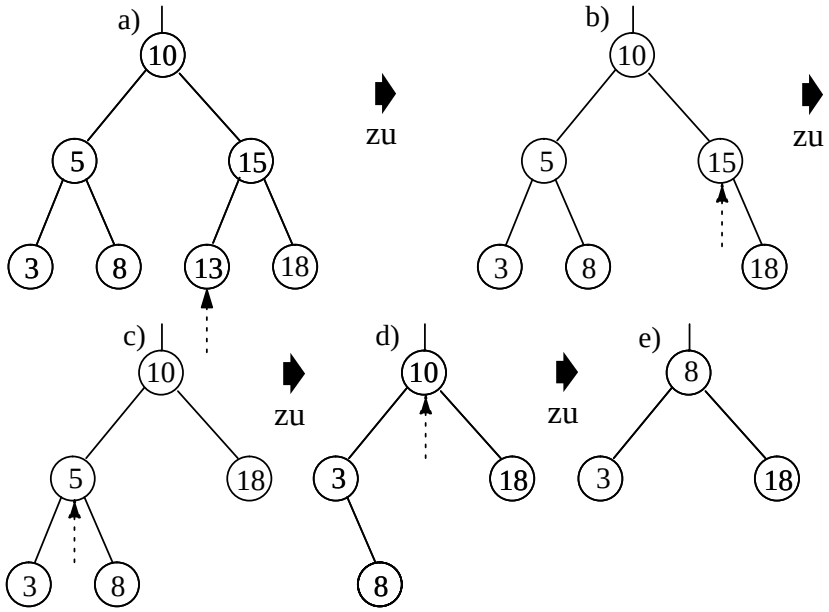


2 Mögl.: a) k ist gößtes El. aus T_1
dann ist $T_2' = T_2$

b) k ist kleinstes El. aus T_2
dann ist $T_1' = T_1$

kleinstes El. aus T_2 oder größtes aus T_1 haben höchstens einen Nachfolger.

– Beispiele

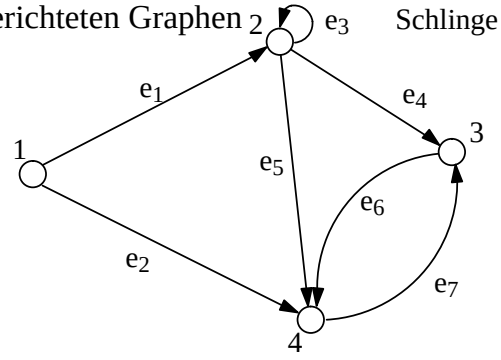


– Einfügen problemlos
  unter bisheriges Blatt
  unter Knoten mit bisher nur anderem Nachfolger
  erzeugt ggfs. Zwischenknoten mit nur einem Nachf.

Graphen und Graphprobleme

Definitionen und Beispiele

– Beispiel eines gerichteten Graphen



G

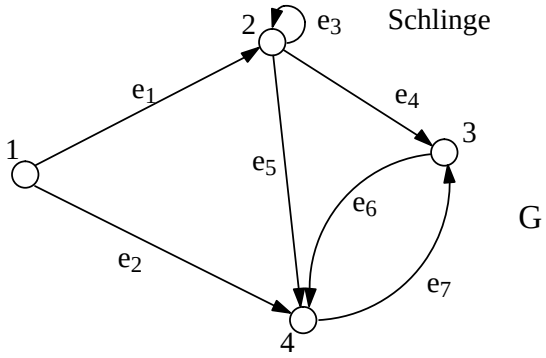
K={ 1, 2, 3, 4 }
E={ (1, 2), (1, 4), (2, 2), (2, 3), (2, 4), (3, 4), (4, 3) }
e_1 e_2 e_3 e_4 e_5 e_6 e_7

– Def.
k1, k_2 *benachbart* $k_1 \vdash\dashv k_2$ g.d.w. $(k_1, k_2) \in E$ oder $(k_2, k_1) \in E$
Nachbarknoten NeighN(k) := {k' | k' ∈ K ∧ k ⊢⊣ k'}
$NeighN^-$(k) Nachbarn über einl. Kanten
$NeighN^+$(k) Nachbarn über ausl. Kanten
inzidierende Kanten IncE(k):={ e | e ∈ E ∧ ∃ k' ∈ K
(e= (k, k') ∨ (k', k)) }
$IncE^-$(k) einl. Kanten
$IncE^+$(k) ausl. Kanten
(Kanten)Grad von k (abgek. deg(k)) ist | IncE(k) |

– Beispiel

In G gilt

$2 \vdash\dashv 3$
$\text{NeighN}(2) = \{1, 2, 3, 4\}$
$\text{NeighN}^{-}(4) = \{1, 2, 3\}$
$\text{IncE}^{-}(3) = \{e_4, e_7\}$
$\deg(2) = 4$



G

– Def. *markierter Graph* $G = (K, E, f_{nl}, f_{el})$ mit
$G' = (K, E)$ ist Graph
$f_{nl} : K \mapsto NL$ *Knotenmark.*, f_{nl} *Knotenmarkierungsfunktion*
$f_{el} : K \mapsto EL$ *Kantenmark.*, f_{el} *Kantenenmarkierungsfunktion*

– Probleme, die auf Graphenprobleme zurückgeführt werden können:
(Knoten: Kreuzungen, Endpunkte von Straßen, Plätze
Kanten: Straßen
Stadtplan, Landkarte → markierter Graph)

Erreichbarkeitsproblem: Ist Stelle 2 von Stelle 1 aus erreichbar?
Kürzester Weg: Welcher ist der beste Weg?
Mehrf. Zusammenhang: Gibt es eine Verbindung, wenn eine Straße/ ein Platz blockiert ist?
Fluß: Was ist die Gesamtkapazität des Straßennetzes zwischen zwei Punkten
Minimalgerüst: Errichtung einer Stromversorgung z. B. für Bauernhöfe

– Bem.: Graphen zur Codierung von Sachverhalten
Markierungen können sein:
Knoten: Klasseneint. f. Objekte: NL endl. nichtl. Menge
Kanten: Klasseneint. f. Bez.: EL endl. nichtl. Menge
Knoten, Kanten: Gewichtungswerte, Kapazitäten

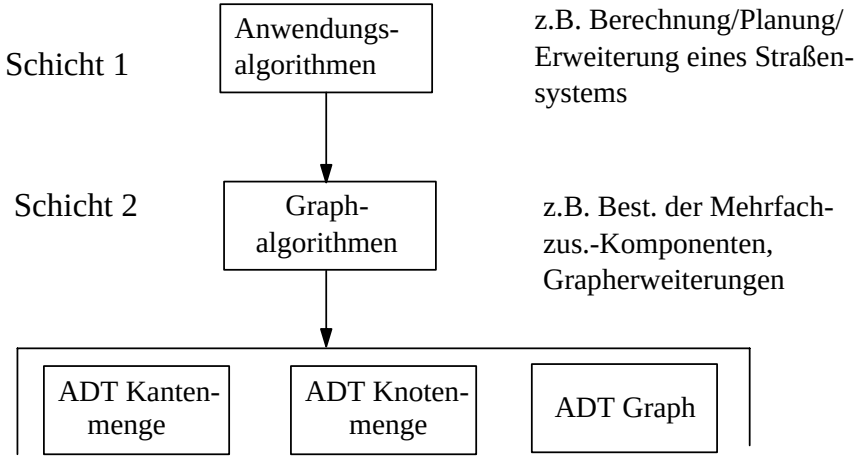
evtl. beides:
Knoten: Menschen,
Klasseneint.: Angestellter, ltd. Angestellter etc
Gewichtung z.B. Alter, Vermögen etc.

– Unterscheidung: Bezeichnung – Markierung

– ADT Graph hätte z.B. folgende Operationen

```
Label(n)
MakeNull()
isEmpty()
OutN(n, elab, NSet)
InN(n, elab, NSet)
OutE(n, ESet)
InE(n, ESet)
.
.
.
```

– daurauf aufsetzend könnten oben skizzierte Probleme programmiert werden



Vorteil: Wären in Schicht 1 und insb. Schicht 2 unabh. von spez. Real. der Graphen und der elem. Graphveränderungen

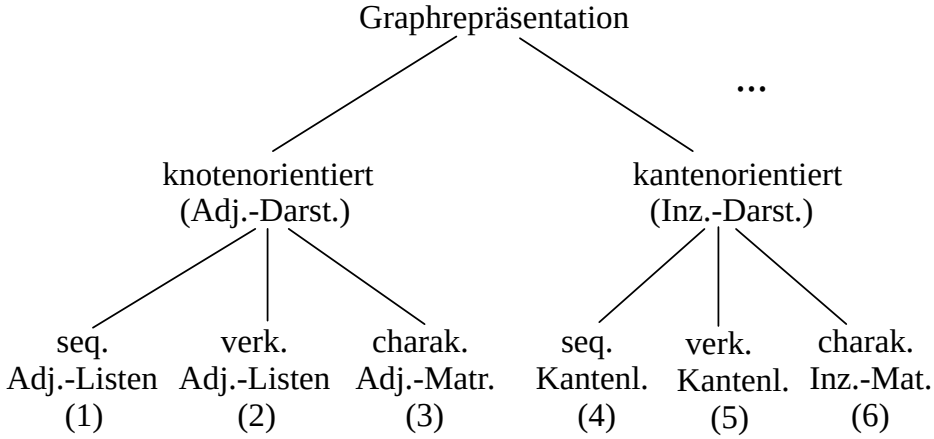
ADT Graph Realisierungsübersicht

Graphrealisierungen

– Repräsentation (Realisierungen) von Graphen
- Zurückführung auf Mengen
  knotenorientiert (adjazenzorientiert):
  Für jeden Knoten: NeighN(k)
  kantenorientiert (inzidenzorientiert):
  Für jede Kante: Anfangsknoten, Endknoten
  ...

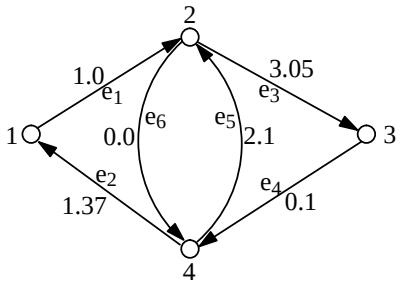
- mögliche Realisierungen der Mengen
  sequentielle Speicherung
  verkettete Speicherung
  (Suchbaumrealisierung)
  charakteristische Speicherung

– somit mögliche Realisierungen



– Auswahl der Speicherung hängt von Operationen ab:
  z.B. sequentielle Kantenliste (Mögl. (4))
  geeignet für Graphen mit beschränkter Kantenzahl, ohne isol. Knoten

1	2	3	4	5	6	
1	4	2	3	4	2	Anfangsknoten
2	1	3	4	2	4	Endknoten
1.0	1.37	3.05	0.1	2.1	0.0	Wert d. Kante



Verwendung der Graphschnittstelle

– Einige typ. Graphoperationen (Pseudocode)
  Wollen für alle Darstellungen folg. Operationen betrachten

  - **for all e in $IncE^+(k)$ do**
    bearbeite alle aus k hinausg. Kanten
  - **for all e in E do**
    bearbeite alle Kanten
  - **isEdge(k1, k2)**

  dabei sei stets n Anzahl der Knoten
  m Anzahl der Kanten

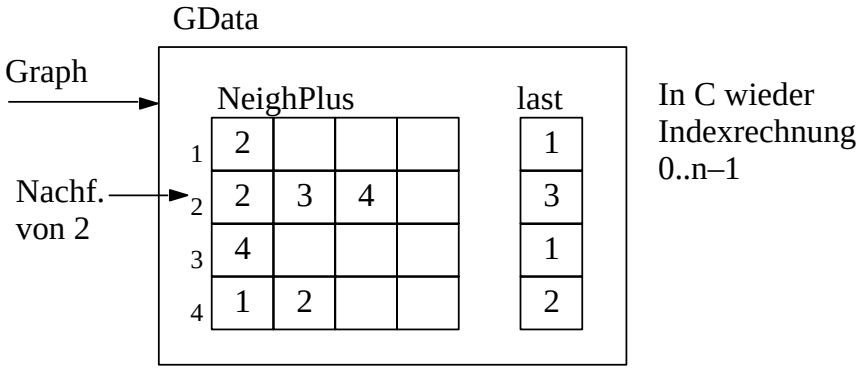
  Knotenbezeichnungen ganzzahlig

Knotenorientierte Realisierungen

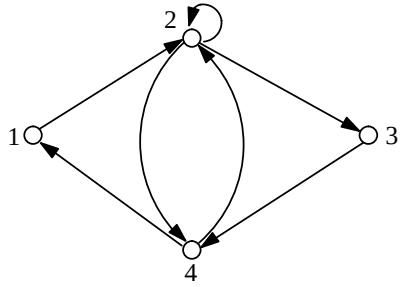
– Bem.
Zunächst ohne Markierungen
Graph völlig durch Adjazenzangaben bestimmt
(genügt bereits $NeighN^+(k)$ für alle $k \in K$)

Seq. Adjazenzlisten

– Bsp.:



In C wieder
Indexrechnung
0..n–1



- Realisierung

```
struct GData {
   int NeighPlus[n][n];
   int last[n];
};
typedef GData* Graph;
```

$O(n^2)$

for all e in $IncE^+(k)$ do

```
   for (int j=0; j<=last[k]; j++) {
      nimm e als (k, NeighPlus[k][j]);
      bearbeite e;
   };
```

$O(|IncE+(k)|)$

for all e in E do

```
   for(int i=0; i<n; i++) {
      for(int j=0; j<=last[i]; j++) {
         nimm e als (i, NeighPlus[i][j]);
         bearbeite e;
      };
   };
```

$O(m)$

isEdge(k_1, k_2)

```
   j=0;
   while (j<=last[k1]) {
      if (NeighPlus[k1][j]==k2) {
         return true;
      };
      j++;
   };
   return false;
```

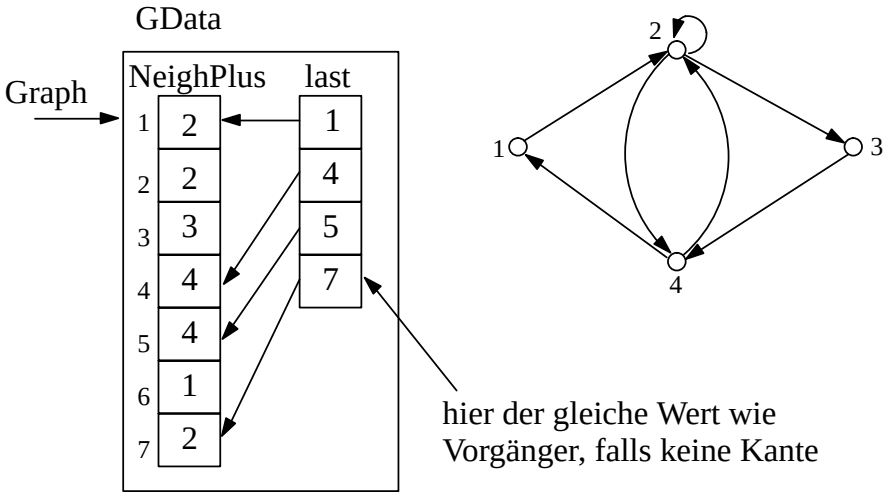
$O(|IncE+(k)|)$

- Wertung
  Hinzunahme von Kanten leicht
  Platzverschwendung enorm: der vollst. Graph tritt selten auf.
  Hinzunahme neuer Knoten schwer

Verdichtete seq. Speicherung

– Bem.:
  Falls m << n^2 oben viel Platz verschenkt.
  Hintereinanderlegen der seq. Knotenliste (Zeilen der Matrix)

– Bsp.



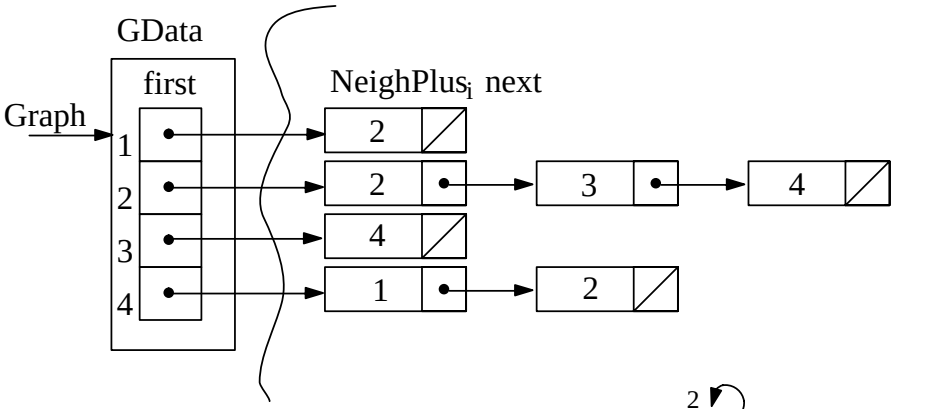
Verkettete Adjazenzlisten

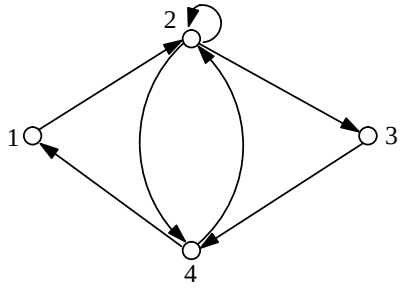
– Bem.
  obg. Realisierungen durch seq. Adjazenzlisten
  (verd. oder unverd.),
  nicht geeignet für Graphen, die sich verändern:
    Knoten einfügen/löschen
    Kanten einfügen/löschen

  Verkettung löst dieses Problem
    Realisierung durch C++ - Zeiger und Haldenobjekte
    Realisierung durch Indexverkettung

– Betrachten verzeigerte Realisierung zunächst ohne Knoten- und Kantenmarkierungen

– Bsp.





first[i] zeigt auf den Anfang der Nachfolgerliste zu Knoten i.

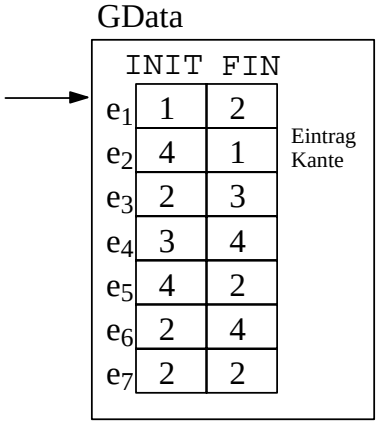
Lösung noch inflexibel: Knoten einfügen und löschen
⇒ Ersetzen first-Feld ebenfalls durch Haldenstruktur

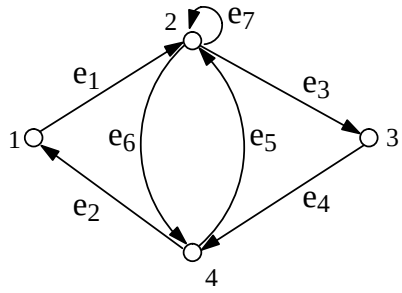
Kantenorientierte Realisierung

– Bemerkungen
synonym inzidenzorientierte Realisierung,
falls isolierte Knoten auftreten, zus. Abspeicherung der Knoten.

Sequentielle Kantenliste

– Beispiel





Darstellung: Quellknoten, Zielknoten zu Kante,
Kantenreihenfolge bedeutungslos

- Realisierung

```
struct KantenEl {
   int Quelle;
   int Ziel;
};
typedef KantenEl GData[m];
typedef GData* Graph;
```

O(m)

for all e in $IncE^+(k)$ do

```
  for (int i=0; i<m; i++) {
     if (*G[i].Quelle==k) {
        e = (k, *G[i].Ziel);
        bearbeite e;
     };
  };
```

O(m)

for all e in E do

```
   analog
```

O(m)

isEdge (k1, k2)

```
  i=0;
  while (i<m) {
     if ((*G[i].Quelle==k1)
          &&(*G[i].Ziel==k2)){
        return true;
     };
     i++;
  };
  return false;
```

O(m)

6 Objektorientierung und weitere Konzepte

Lernziele:

Klassenhierarchien durch Generalisierung/Spezialisierung
Verarbeiten verschiedener Objekte mit dyn. Bindung
virtuelle Methoden, abstrakte Klassen
Generizität und Templates

...

Vererbung zwischen Klassen

ADTs und Klassen

– Datenabstraktion
  ADO simuliert durch Header- und Implementation-File
    als “Modul”
  ADT als Klasse in C++
    Schnittstelle muß nicht simuliert werden
    Rumpf wird “simuliert”

– Schnittstelle von Klassen enthält
  public-Teil: für alle potentiellen Klienten
  private-Teil: nur für die Klassenimplementation
  protected-Teil: für spezialisierte Klassen (s.u.)

– kein Typbezeichner wird exportiert
  Klasse = Typ
  kein formaler Parameter für das gesamte Objekt
  `this` im Rumpf kennzeichnet das nicht vorhandene
    Formalparameterobjekt

– Begriffe der objektorientierten Programmierung
  Methode: (Realisierung der) Zugriffsoperation
  Botschaft: Aufruf einer Zugriffsoperation/Methodenaufruf
  Punktnotation (oder Pfeil) für Botschaft an Objekt

Datentypmodule als Klassen in C++

```
// angestellter.h
#ifndef __INC_ANGESTELLTER_H__
#define __INC_ANGESTELLTER_H__

class Angestellter {
public:
   Angestellter (char *name, int tarif);
   // ...
   char *gibName();
   float gehalt();
   // ...

private:
   char *name;
   int tarifgruppe;
};
#endif // __INC_ANGESTELLTER_H__

// Class Body Angestellter
#include ”angestellter.h”

Angestellter::Angestellter(char *name,
                           int tarif) {
   this->name = name;
   tarifgruppe = tarif;
}

...

// End Body Angestellter
```

Oberklasse-Beispiel

```
//personal.h
#ifndef __INC_PERSONAL_H__
#define __INC_PERSONAL_H__

class Personal {
public:
   Personal(char *name);
   // ...
   char *gibNamen();
   virtual float gehalt();

private:
   char *name;
   // ...
};
#endif // __INC_PERSONAL_H__


// Class Body Personal
#include "personal.h"

Personal::Personal(char *name) {
   this->name = name;
}

char Personal::gibNamen() {
   return name;
}

float Personal::gehalt() {
   return 500.0; // Festbetrag
}

// End Body Personal
```


Unterklasse-Beispiel

```
#ifndef __INC_ANGESTELLTER_H__
#define __INC_ANGESTELLTER_H__
// Import Personal
#include "personal.h"

class Angestellter:public Personal {
public:
   Angestellter(char *name, int tarif);
   // ...
   virtual float gehalt();
   // ...
private:
   // Anreicherungen gegenueber Personal
   int tarifgruppe;
};
#endif

// Class Body Angestellter
// Import Personal
#include "personal.h"
// FROM Tarife IMPORT TarifFaktor, TarifTab
#include "tarife.h"
#include "angestellter.h"

Angestellter::Angestellter (char *name,
              int tarif):Personal(name) {
   tarifgruppe = tarif;
}

// Gehaltsberechnung für Angestellter
float Angestellter::gehalt(void) {
   return (TarifFaktor*TarifTab[tarifgruppe]
      + Personal(*this).gehalt());
}
// End Body Angestellter
```

Weitere Spezialisierung

```
#ifndef __INC_LEITENDERANGESTELLTER_H__
#define __INC_LEITENDERANGESTELLTER_H__
//Import Angestellter
#include "angestellter.h"

class LeitenderAngestellter:
          public Angestellter {
public:
   LeitenderAngestellter(char* name, int tarif
                         float zul);
   float gehalt();

private:
   // ...
   float zulage;
   // ...
};
#endif

// Class Body LeitenderAngestellter
// Import Angestellter
#include "angestellter.h"
#include "LeitenderAngestellter.h"

LeitenderAngestellter::LeitenderAngestellter
      (char *name, int tarif,
       float zul):Angestellter(name, tarif) {
   zulage = zul;
}

// Gehaltsabrechnung fuer leitende Angestellte
float LeitenderAngestellter::gehalt() {
  return zulage+Angestellter(*this).gehalt();
}
// End Body LeitenderAngestellter
```


Idee der Vererbung, Nutzung, Gefahren

– Vererben von Eigenschaften, Hinzufügen spezieller Eigenschaften: Datenaufbau und Methoden.
Oberklasse, Unterklasse: Oberklasse vererbt, Unterklasse erbt; Unterklasse ist "abgeleitete" Klasse.

– Vererbungshierarchie wichtige Struktureigenschaft von Systemen:
Einfachvererbung, Mehrfachvererbung, Spezialisierung, Generalisierung

– zur Wiederverwendung von Eigenschaften (Schnittstelle)
zur Wiederverwendung von Code (Gehaltsberechnung)

– Gefahren der OOP
Klassen für alles (nicht nur für abstrakte Datentypen)
Wilde Beziehungen zwischen Klassen
Umstrukturierung von Vererbungshierarchien schwierig
dynamisches Binden erzeugt bei nichtdisziplinierter Programmierung unzuverlässige Programme

Nutzung von Vererbung

```
// FROM iostream IMPORT cout, <<
#include <iostream.h>
// Import Angestellter
#include "angestellter.h"
// Import LeitenderAngestellter
#include "leitenderangestellter.h"

void main () {
   Personal *a =
       new LeitenderAngestellter("Meyer",
                                 4, 2000);
   Personal *b =
       new Angestellter("Mueller", 2);
   Personal *c;

   cout << a->gehalt() << endl; // mit Zulage
      // fuehrt zu Aufruf der speziellen
      // Methode unter Nutzung der Methode der
      // Oberklasse, die wiederum die Methode
      // von Personal benutzt
   cout << b->gehalt() << endl; // ohne Zulage
   c = Angestellter(*a);
   cout << c->gehalt() << endl; // mit Zulage
}
```

Gemeinsame Verarbeitung und dyn. Bindung

Dispatching

– Jedes Objekt eines Typs aus einer Klassenhierarchie hat best. Typ während gesamter Existenz

– gemeinsame Verarbeitung ohne Fallunterscheidung
  s. obiges Beispiel
  Gehaltsberechnung je nach Typ angesprochen,
  macht jeweils etwas anderes

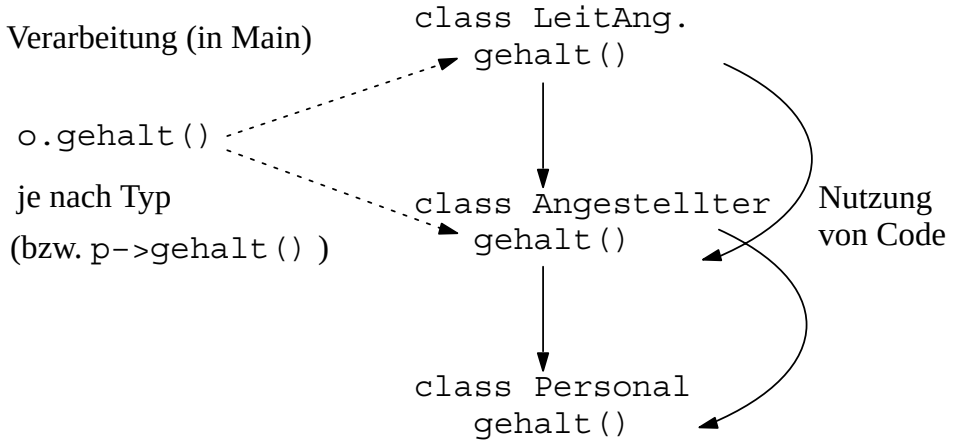
– Anspringen der spezifischen Gehaltsberechnung aufgrund einer Laufzeitkennung für Typ ("tag")

Verarbeitung von Objekten aus einer Vererbungshierarchie

geg. Vererbungshierarchie

```
typedef Personal *PersZeiger;

void GehaltsBestimmung(Buffer *b) {
   while (b->nonempty()) {
      PersZeiger p;
      b->dequeue(p);
      cout << "Name: " << p->gibNamen();
      // ruft gibNamen() von Personal auf
      cout << "   Gehalt: " << p->gehalt();
      // Dispatching: gehalt() von Personal,
      // Angestellter oder Leitender-
      // Angestellter wird aufgerufen.
      cout << endl;
   }
}

class Buffer { // zur Ablage von Personen
public:        // aus Vererbungshierarchie
   Buffer();
   void enqueue(PersZeiger x);
   void dequeue(PersZeiger &x);
   bool nonempty();
   bool nonfull();
private:
   // ...
   Personal *bc[N]
};

// class body
   ...
   // z.B. zirkulaere Speicherung;
   // man beachte, dass das zirk. Feld jetzt
   // nur Zeiger auf Personen enthaelt
// end class body
```

Dispatching (dyn. Bindung zur Laufzeit je nach Typ)



– Voraussetzung:
gleicher Name für Operationen zu Objekten verschiedenen Typs in verschiedenen Klassen

Name an der Wurzelklasse der Teilhierarchie (hier `Personal`) mit `virtual` gekennzeichnet

Sichtbarkeit für Klassen

– Sichtbarkeit von Teilen der Exportschnittstelle
  private: nur im Rumpf der Klasse
  public: für alle Klienten der Klasse
  protected: nur für Spezialisierungen

– Sichtbarkeit der gesamten Schnittstelle
  Vererbung mit public: auch weitere Spezialisierungen können (nach Konvertierung) Operationen nutzen
  Vererbung ohne public ist private: Die Schnittstelle ist nur im Rumpf der Spezialisierung (nach Konvertierung) nutzbar

Virtuelle Methoden, abstrakte Klassen

– Vererbungshierarchien arbeitsteilig erstellt
  gemeinsames Protokoll bei Wurzelklasse festlegen
  abstrakte Klasse (hat keine Objekte)
  Achtung: DA $\neq$ abstr. Klasse

– gemeinsames Protokoll
  Zur Handhabung der Personen eines Unternehmens
  Methoden: keine Mischung abstr. und konkreter Methoden

Einführung einer abstrakten Klasse

– keine Komponenten
  nur pure virtual Methoden

```
class PersProtokol {
public:
   virtual char *gibNamen() = 0;
   virtual float gehalt() = 0;
   ...
};


class Personal : public PersonalProtokoll {
public:
   Personal(char *n);
   char *gibNamen();
   ...
   virtual float gehalt();

private:
   char *name;
};


// Class body

float Personal::gehalt() {
   return 500.0; // Festbetrag
}
...

// End Class body
```


Generizität und Templates

Parametrisierung für Wiederverwendung

– Parameter für
  verschiedene Typen
  Funktionen mit best. Profil
  Dimensionierung

– folgendes Beispiel generischer Puffer
  Eintragstyp variabel
  und Behältergröße einstellbar

```
template<class T, int size>
class Buffer {
public:
  Buffer();
  void enqueue(T x);
  void dequeue(T &x);
  bool nonempty();
  bool nonfull();

private:
  // ...
  T bc[size];
};

// Generic Class Body
template<class T, int size>
Buffer<T, size>::Buffer() {
  // ...
}

template<class T, int size>
void Buffer<T, size>::enqueue(T x) {
  // ...
} // End Body
```

```
int main()
{
  int i;
  float f;

  Buffer<int, 3> intbuffer;
  Buffer<float, 2> floatbuffer;

  intbuffer.enqueue(1);
  intbuffer.enqueue(2);
  intbuffer.enqueue(3);
  floatbuffer.enqueue(1.111);
  floatbuffer.enqueue(2.222);

  intbuffer.dequeue(i); cout << i << endl;
  intbuffer.dequeue(i); cout << i << endl;
  intbuffer.dequeue(i); cout << i << endl;
  floatbuffer.dequeue(f); cout << f << endl;
  floatbuffer.dequeue(f); cout << f << endl;
  return 0;
}


/* Ausgabe:
1
2
3
1.111
2.222
*/
```

– heterogene Kollektion
  Kollektion für Objekte versch. Typs
  Typ aus Vererbungshierarchie
  Kollektion für Zeiger auf versch. Objekte

– Bsp. Puffer von vorne
  jetzt für versch. Vererbungshierarchien anwendbar

– Anwendbar für Vererbungsh. `PersProtokoll`
  oder etwa für graphische Objekte

```
template<class T, int size>
 void GehaltsBestimmung(Buffer<T, size>* b){
  while (b->nonempty()) {
    PersZeiger p;
    b->dequeue(p);
    cout << "Name: " << p->gibNamen();
    // ruft gibNamen() von Personal auf
    cout << "   Gehalt: " << p->gehalt();
    cout << endl;
    // Dispatching: gehalt() von Personal,
    // Angestellter oder Leitender-
    // Angestellter wird aufgerufen.
  }
}

int main()
{
  Personal *p=new Personal("Willi Wacker");
  Angestellter *a=
        new Angestellter("Ekel Alfred", 1);
            // Tarif 1
  LeitenderAngestellter *la=
        new LeitenderAngestellter
                ("Charlies Papa", 2, 100);
       // Tarif 2, Zulage 100

  Buffer<PersZeiger, 3>* personenbuffer =
        new Buffer<PersZeiger, 3>;

  personenbuffer->enqueue(p);
  personenbuffer->enqueue(a);
  personenbuffer->enqueue(la);
  GehaltsBestimmung( personenbuffer );
  return 0;
}
```

Ausnahmebehandlung (Exception handling)

Allgemeines

– Laufzeitfehler führen zu schwerwiegenden Störungen,
  oft nicht an der Stelle ihres Auftretens sondern später
  schwer zu erkennen

– Alle Fehlersituationen von Code abzufangen
  ist mühselig, oft gar nicht möglich

– Ausnahmebehandlung als Sprachkonzept
  - Fehlerdefinition
  - Fehlerbehandlung
  - geregeltes Programmbeenden

– Trennung von
  - normalem Ablauf
  - und Fehlerbehandlung

– Fehler allgemeiner zu "Ausnahmen"
  - für alle außergewöhnlichen Situationen
  - außerhalb des normalen Ablaufs

– leider übl. Fehlersituationen
  - Bereichsüberschreitungen bei Feldern
  - Overflow, Underflow num. Datentypen
  - nicht mit autom. Ausnahmeerweckung verbunden

OO und Ausnahmedefinition

– Für Fehler Hierarchie aus einfachen Klassen
  haben entweder keine Struktur (meist)
  oder int. Struktur dient zur Beschreibung des Fehlers

– Hierarchie vorab und global für Projekt festlegen:

```
class DataExc {
};

class Overflow  : public DataExc {
};
class Underflow : public DataExc {
};

...
```

– alternativ, weniger gut
  Ausnahmen in Baustein festlegen und exportieren

– Fehler sind Objekte zu Fehlerklassen

Auslösen (Werfen) von Fehlern

```
// Fehlerklassen seien sichtbar

class Buffer {
  // Schnittstelle wie sonst, aber festl.,
  // welche Operationen welche Ausnahmen
  // auslösen können
  // ...
public:
   void enqueue(PersZeiger x);  // Overflow
   void dequeue(PersZeiger &x); // Underflow
  // ...
};

// Class Body
// ...

void Buffer::enqueue(PersZeiger x)
{
   if (n < N) {
      ...                  // Normalfall
   } else {
      throw Overflow(); // Ausnahme
   }
}

// dequeue() entsprechend mit throw Underflow

// End Body
```

Reaktion mit Ausnahmebehandler: Ausnahmebehandlung

```
try { // beliebige Anweisungen, die enqueue
      // und dequeue aufrufen
   ...
}
catch (Overflow) {
   // Ausnahmebehandlung durch Ausnahme-
   // behandler
   // irgendwelche Anweisungen zur
   // Begrenzung des Schadens
}
catch (Underflow) {
   // ...
   cout << "Fehler 37: Lesen einer leeren ";
   cout << "Dateiliste." << endl;
}
catch (...) { // irgendeine Ausnahme
   // ...
}
```

Weiterreichen von Ausnahmen

– gleiche Ausnahmen weitergereicht
  explizit durch `throw();`
  implizit, wenn kein passender Ausnahmebehandler vorhanden

– ist kein Ausnahmebehandler vorhanden, so wird das Gesamtprogramm abgebrochen
  ebenso, wenn "außerhalb" einer Ausnahmebehandlung keine mehr stattfindet und eine Ausnahme auftrat

Anhang I:
C++ Syntax

Syntaxnotation

Die im folgenden verwendete Notation zur Beschreibung der Syntax einer Programmiersprache nennt sich EBNF–Notation (Erweiterte Backus–Naur–Form). Diese Form der Syntaxbeschreibung ist weit verbreitet und – nach kurzer Einarbeitung – leicht zu lesen und zu verstehen. Im folgenden wird eine Variante von EBNF eingeführt; es gibt viele Varianten. Eine Syntaxbeschreibung (Grammatik) in EBNF besteht aus einer Menge von Regeln. Regeln haben eine linke und eine rechte Seite, die durch die Zeichen ::= getrennt sind. Auf der linken Seite steht ein sogenanntes Nichtterminalsymbol. Ein Nichtterminalsymbol faßt einen bestimmten Teil einer Sprache zusammen, z.B. gibt es in der C++–Grammatik das Nichtterminalsymbol statement, das für eine Anweisung in einem C++–Programm steht. Die Bedeutung ist, daß an all den Stellen, an denen in der Grammatik das Nichtterminal statement verwendet wird, eine Anweisung in einem Programm stehen kann.

Auf der rechten Seite einer Regel steht, wie der Teil der Sprache für das Nichtterminalsymbol auf der linken Regelseite aufgebaut ist. Dazu werden wiederum andere Nichtterminalsymbol und auch Terminalsymbole der Grammatik verwendet. Terminalsymbole sind Zeichen oder Zeichenfolgen, die in einem Programm genau so auftauchen, wie sie in der Grammatik stehen. Um Terminalsymbole von Nichtterminalsymbolen besser unterscheiden zu können, werden sie in einer anderen Schriftart dargestellt. Bei uns ist das der Zeichensatz `Courier`, während Nichtterminalsymbole in Helvetica gesetzt sind.

Auf der rechten Seite einer Regel werden Terminal– und Nichtterminalsymbole miteinander verknüpft. In der regulären EBNF gibt es fünf verschiedene Arten der Verknüpfung:

1. Sequenz: Durch Hintereinanderschreiben von Zeichen und Wörtern gibt man an, daß diese Teile der Sprache auch in einem Programm hintereinander auftauchen.
2. Alternative: Mit dem Zeichen | werden Alternativen gekennzeichnet. Taucht in einer Regeln zum Beispiel `a` | `b` auf, dann steht in einem Programm entweder `a` oder `b`, aber nicht beides.
3. Option: Durch Klammerung mit eckigen Klammern [und] gibt man an, daß der Teil der Regel in Klammern optional ist. Er kann in einem Programm an der Stelle stehen, muß aber nicht.
4. Wiederholung: Die Klammerung mit geschweiften Klammern { und } gibt an, daß der Teil der Beschreibung in den Klammern an dieser Stelle null bis beliebig oft mal wiederholt werden kann.
5. Nichtleere Wiederholung: Durch Nachstellen eines $^+$ geben wir bei der Wiederholung an, daß der Inhalt der Klammern mindestens einmal im Programmtext auftauchen soll.
6. Gruppierung: Mit runden Klammern (und) werden Gruppen gebildet, wie man das vom Umgang mit Formeln gewohnt ist.
7. Rekursion: Die Rekursion wird nicht besonders gekennzeichnet. Bei der Rekursion werden Nichtterminalsymbole verwendet, deren rechte Regelseiten direkt oder indirekt wiederum das Nichtterminalsymbol verwenden, zu dessen Erklärung sie gerade benutzt werden. Ein Beispiel dafür geben wir unten an.

C++–Grammatik

Die folgende Grammatik gibt nicht den gesamten C++–Sprachumfang wieder. Sie dient aber als Grundlage für die in der Vorlesung verwendete Teilmenge von C++. An vielen Stellen ist die Grammatik restriktiver als es C++ zuläßt. So haben wir insbesondere dem Anweisungsteil eine Struktur aufgeprägt, wie sie auch in anderen Programmiersprachen zu finden ist. Die Art, wie die rechte Regelseite aufgeschrieben ist, gibt in der Regel an, wie wir uns das Layout des entsprechenden Programmteils vorstellen.

Schlüsselwörter

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

Lexikalische Syntax

identifier ::= letter { nondigit | digit }

literal ::= integer_literal | character_literal | floating_literal
| string_literal | boolean_literal

integer_literal ::= dec_int_literal [integer_suffix]
| hex_int_literal [integer_suffix]
| oct_int_literal [integer_suffix]

dec_int_literal ::= nonzero_digit { digit }

hex_int_literal ::= `0x`{hexadecimal_digit}$^+$

oct_int_literal ::= `0` { octal_digit }

integer_suffix ::= [`l` | `L`][`u` | `U`]

letter ::= `a` | `b` | `c` | `d` | `e` | `f` | `g` | `h` | `i` | `j`
| `k` | `l` | `m` | `n` | `o` | `p` | `q` | `r` | `s` | `t` | `u`
| `v` | `w` | `x` | `y` | `z` | `A` | `B` | `C` | `D` | `E` | `F`
| `G` | `H` | `I` | `J` | `K` | `L` | `M` | `N` | `O` | `P` | `Q`
| `R` | `S` | `T` | `U` | `V` | `W` | `X` | `Y` | `Z`

nondigit ::= _ | letter

digit ::= `0` | non_zero_digit

non_zero_digit ::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

octal_digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`

hexadecimal_digit ::= digit | `a` | `b` | `c` | `d` | `e` | `f`
| `A` | `B` | `C` | `D` | `E` | `F`

character_literal ::= c_char

floating_literal ::= {digit}$^+$.{digit}$^+$ [`e` [sign] {digit}$^+$] [floating_suffix]

sign ::= `+` | `-`

floating_suffix ::= `f` | `F` | `l` | `L`

string_literal ::= ″{s_char}″

s_char ::= graphic_char außer \, ″ und eol | escape_sequence

c_char ::= graphic_char außer \, und eol | escape_sequence

graphic_char ::= Die Menge der darstellbaren Zeichen

escape_sequence ::= simple_esc_seq | oct_esc_seq | hex_esc_seq

simple_esc_seq ::= `\″` | `\?` | `\\` | `\a` | `\b` | `\f` | `\n` | `\r` | `\t` | `\v`

oct_esc_seq ::= `\` octal_digit octal_digit octal_digit

hex_esc_sequence ::= `\x` { hexadecimal_digit }$^+$

boolean_literal ::= **`true`** | **`false`**

Ausdrücke

expression ::= relation_expression { logical_op relation_expression }

logical_op ::= `||` | `&&`

relation_expression ::= bit_expression [relation bit_expression]

relation ::= `==` | `!=` | `<` | `>` | `<=` | `>=`

bit_expression ::= add_expression { bit_op add_expression }

bit_op ::= `&` | `^` | `|` | `<<` | `>>`

add_expression ::= mult_expression { add_op mult_expression }

add_op ::= `+` | `-`

mult_expression ::= unary_expression { mult_op unary_expression }

mult_op ::= `*` | `/` | `%`

unary_expression ::= [unary_operator] simple_expression
| new_expression | delete_expression

unary_operator ::= `*` | `&` | `+` | `-` | `!` | `~`

new_expression ::= **`new`** identifier { `[` expression `]` } [`(` expression_list `)`]

expression_list ::= expression {`,` expression}

delete_expression ::= **`delete`** [`[]`] simple_expression

simple_expression ::= primary_expression | cast_expression | typeid_expression

primary_expression ::= literal | **`this`** | `(` expression `)` | qualified_name

qualified_name ::= {*class_or_namespace*_identifier ::} name

name ::= identifier | operator_symbol | indexed_component
| selected_component | function_call

operator_symbol ::= **`operator`** operator

operator ::= logical_op | bit_op | relation | add_op | mult_op

indexed_component ::= name `[` expression `]`

selected_component ::= name `.` identifier | name `->` identifier

function_call ::= name `(` [expression_list] `)`

cast_expression ::= **`dynamic_cast`** `<` *type*_identifier `>` `(` expression `)`
| **`static_cast`** `<` *type*_identifier `>` `(` expression `)`
| **`reinterpret_cast`** `<`*type*_identifier`>` `(`expression`)`
| **`const_cast`** `<` *type*_identifier `>` `(` expression `)`

typeid_expression ::= **`typeid`** `(` expression `)`

Anweisungen

statement_list ::= { statement`;` }$^+$

statement ::= [identifier `:`] simple_statement
| [identifier `:`] compound_statement

simple_statement ::= assignment_statement | jump_statement
| throw_statement | inc_statement
| dec_statement | call_statement

compound_statement ::= block | if_statement | switch_statement | while_statement
| do_while_statement | for_statement | try_block

assignment_statement ::= identifier assignment_operator expression

assignment_operator ::= `=` | `*=` | `/=` | `%=` | `+=` | `-=` | `>>=` | `<<=` | `&=`
| `^=` | `|=`

block ::= `{`
declaration_list
statement_list
`}`

if_statement ::= **`if`** `(` expression `)` `{`
statement_list`}`
{**`else if`** `(` expression `)` `{`
statement_list `}` **`}`**
[**`else`** `{`
statement_list
`}`]

switch_statement ::= **`switch`** `(` expression `)` `{`
{case_list statement_list **`break;`**}
[**`default:`** statement_list]
`}`

case_list ::= { **`case`** *const*_expression **`:`** }$^+$

while_statement ::= **`while`** `(` expression `)` `{`
statement_list
`}`

do_while_statement ::= **`do`** `{`
statement_list `}`
`while` `(` expression `)`

for_statement ::= **`for`** `(` *discrete*_init_decl `;` condition `;` inc_dec `)` `{`
statement_list
`}`

init_decl ::= *type*_identifier init_object

inc_dec ::= inc_statement | dec_statement

jump_statement ::= **`break`** | **`continue`** | **`return`** [expression]
| **`goto`** *label*_identifier

try_block ::= **`try`** block handler_sequence

handler_sequence ::= {**`catch`** `(` exception_declaration `)` `{`
statement_list
`}` }$^+$

throw_statement ::= **`throw`** [expression]

inc_statement ::= designator`++`

dec_statement ::= designator`--`

call_statement ::= name `(` expression_list `)`

Deklarationen

declaration_list ::= { declaration `;` }

declaration ::= object_declaration | type_declaratoin
| subprogram_decl | subprogram_definition
| linkage_specification

object_declaration ::= [storage_class] *type*_identifier [cv_spec] init_object_list

storage_class ::= **`auto`** | **`register`** | **`static`** | **`extern`** | **`mutable`**

cv_spec ::= { **`const`** | **`volatile`** }+

init_object_list ::= init_object { `,` init_object }

init_object ::= [`*` | `&`] identifier [`[` *constant*_expression `]`] [initializer]

initializer ::= `=` init_expression | `(` expression_list `)`

init_expression ::= expression | `{` [init_expression_list] `}`

init_expression_list ::= init_expression { `,` init_expression }

type_declaration ::= class_declaration | struct_declaration | union_declaration
| enum_declaration | typedef_declaration

class_declaration ::= **`class`** identifier [inheritance_spec]
{
[**`public`**`:` decration_list]
[**`protected`**`:` declaration_list]
[**`private`**`:` declaration_list]
}

inheritance_spec ::= `:` inheritance { `,` inheritance }

inheritance ::= [**`virtual`**] [access_specification] *class*_identifier

access_specification ::= **`public`** | **`protected`** | **`private`**

struct_declaration ::= **`struct`** identifier {
{ object_declaration `;` }
}

union_declaration ::= **`union`** identifier {
{ object_declaration `;` }
}

enum_declaration ::= **`enum`** identifier {
enumerator_definition { `,` enumerator_definition}
}

enumerator_definition ::= identifier [`=` *constant*_expression]

typedef_definition ::= **`typedef`** *type*_identifier [`*` | `&`] identifier
[`[` *constant*_expression `]`]

subprogram_decl ::= [subp_spec] (function_decl | procedure_decl)

subp_spec ::= **`inline`** | **`virtual`** | **`explicit`**

function_decl ::= *type*_identifier qualified_name parameter_spec

procedure_decl ::= **`void`** qualified_name parameter_spec

parameter_spec ::= `(` [formal_parameter { `,` formal_parameter }] `)`

formal_parameter ::= *type*_identifier [`*` | `&`] identifier [`=` *constant*_expression]

subprogram_definition ::= subprogram_decl [**`try`**] [ctor_initializer]
block
[handler_sequence]

ctor_initializer ::= `:` member_initializer { `,` member_initializer }

member_initializer ::= identifier `(` [expression_list] `)`

linkage_specification ::= **`extern`** string_literal declaration
| **`extern`** string_literal `{` declaration_list `}`

Programmstruktur

file ::= program_file | header_file

context_clause ::= **`using namespace`** *namespace*_name `;`
| `#include "`*file*_identifier`"`
| `#include <`*file*_identifier`>`

header_file ::= {context_clause} { header_declaration }+

program_file ::= {context_clause} {declaration}+

header_declaration ::= subprogram_decl | *const*_object_declaration
| type_declaration

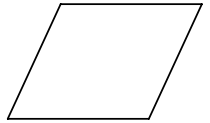
Anhang II:

Graphische Notationen für Programme

Flußdiagramme nach DIN 66001

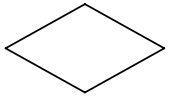
Anfang/Ende:

Ein–/Ausgabe:



andere Anweisung:

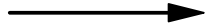
Abfrage:



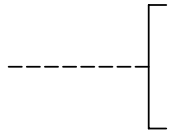
Unterprogramm,
Unterablauf:

Konnektor:

Flußlinie:



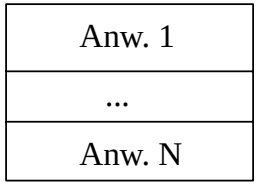
Bemerkung:



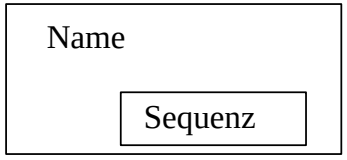
<u>keine Datenstrukturierung</u>
<u>Probleme mit Flußdiagrammen: Spaghettiprogramme</u>

Nassi–Shneiderman–Diagramme
(Struktogramme)

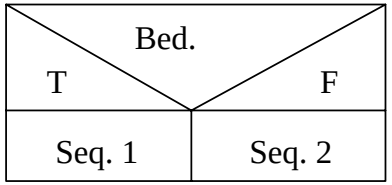
Sequenz:



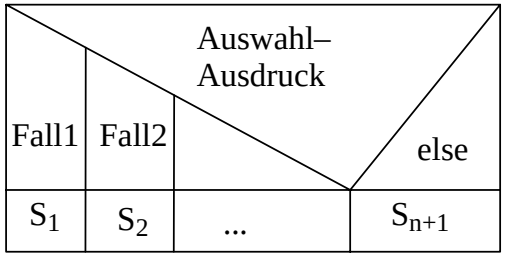
Unterpr.
Unterabl.



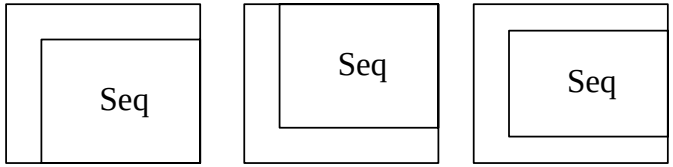
bed. Anw.



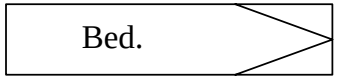
Auswahl–
anw.



Schleifen



kontr. Sprung



<u>ebenfalls keine Datenstrukturierung</u>

Anhang III:

Arbeiten am CIP–Pool in C++

Der CIP–Pool

Im Rahmen der Übungen zur Vorlesung ”Algorithmen und Programmiertechniken” bieten wir Ihnen die Möglichkeit, Übungsaufgaben, bei denen programmiert werden soll, an den Rechnern des CIP–Pools des Lehrstuhls für Allgemeine Elektrotechnik und Datenverarbeitungssysteme zu lösen. Um einen reibungslosen Betrieb der Übungen zu erreichen, bitten wir Sie, die folgenden Punkte zu beachten:

- Die Übungen werden im CIP–Pool von uns montangs zwischen 14:00 und 16:00 Uhr betreut.
- Sie können die Aufgaben auch zu den normalen Öffnungszeiten des CIP–Pools bearbeiten. Dann steht allerdings niemand zu Verfügung, um übungsspezifische Fragen zu beantworten. Bitte stellen Sie solche Fragen nur während der Beratungszeit am Mittwoch oder in Ihrer Übungsgruppe.

Die Rechner im CIP–Pool

Im CIP–Pool stehen etwa 80 PCs zur Verfügung. Alle Geräte bieten die gleichen Voraussetzungen für das Programmieren, da sie vernetzt auf den selben Server zugreifen. Die Rechner sind so konfiguriert, daß nach dem Einschalten ein Auswahlmenü erscheint, das die Wahl zwischen drei Betriebssystemen anbietet: DOS, Windows NT und Linux. Falls noch Windows NT läuft muß es erst heruntergefahren werden (mit `Strg+Alt+Entf`, `Herunterfahren`, `Herunterfahren und neu starten`), bei DOS genügt `Strg+Alt+Entf`. Der Rechner bootet und das Auswahlmenu erscheint. Mit den Pfeiltasten (Cursortasten) wird Linux gewählt und danach mit der Eingabetaste (↵) bestätigt.

Kennungen

Für die Übungen können die Kennungen aus dem AI–Praktikum I verwendet werden. Wer keinen solchen Account hat, kann ihn beim Übungsgruppenleiter oder bei unserer CIP–Beratung beantragen.

Anmelden / Einloggen

Nachdem als Betriebssystem Linux ausgewählt wurde, bootet der Rechner und meldet sich mit einem Text–Login, nach einer kurzen Zeit

auch mit einem graphischen Login. Dort wird die Kennung und das Paßwort eingegeben. Ist die Anmeldung erfolgreich, erscheinen auf dem Bildschirmhintergrund einige Buttons und ein Fenster mit einer Kommandozeileneingabe (Shell). Eine Einführung in X, den Fvwm2 oder Unix kann hier nicht gegeben werden, daher hier nur kurz das Wichtigste.

Wenn Sie auf dem Hintergrund die linke, mittlere oder rechte Maustaste drücken bekommen Sie jeweils verschiedene Menus in denen Sie Programme starten, zu anderen Fenster wechseln und Fensterfunktionen aufrufen können.

Als Windowmanager ist der Fvwm2 eingestellt, mit ’`Andere Fenstermanager – Fvwm95 starten`’ können Sie den Fvwm95 starten, der eine etwas Windows95–ähnlichere Oberfläche bietet. Im graphischen Login können Sie unter ‘Session Type’ auch kde wählen, der moderner daherkommt.

Unter den fvwm–Windowmanagern gibt es einen *Pager* (oben links) mit dem Sie zwischen verschiedenen *virtuellen Bildschirmen* umschalten können (das geht auch über Tastatur mit `Shift+Alt+Cursortaste`). In kde läßt sich das einstellen, defaultmäßig wird mit `Strg–Tab` gewechselt.

Abmelden / Ausloggen

Am Ende einer Sitzung führen Sie bitte folgende Schritte aus, damit der Rechner anschließend in einem vernünftigen Zustand ist:

- Verlassen Sie einen ggf. geöffneten Emacs durch die Tastenkombination `Strg-x Strg-c`. Ggf. fragt der Emacs noch, ob nicht gespeicherte Dateien gespeichert werden sollen, dann schließt sich das Editorfenster.
- Mit `Arbeitsmenu – Fenstermananger – Fvwm2 und X beenden` wird XFree86 beendet.
- Den Rechner kann am Text–Login mit `Strg+Alt+Entf` oder im graphischen Login über den Beenden–Button herunterfahren werden.

`man befehl`	Anzeigen der Hilfeseite zu einem Befehl.
`ls [pfad]`	Anzeigen des Inhalts eines Verzeichnisses.
`cd [pfad]`	Wechseln in ein anderes Verzeichnis. Wird *pfad* weggelassen, wird ins Homeverzeichnis gewechselt.
`cp datei ziel`	Kopieren einer Datei. *ziel* muß angegeben werden und kann ein neuer Dateiname, der Name einer existierenden Datei oder ein Verzeichnis sein.
`mv datei ziel`	Verschieben oder Umbenennen einer Datei.
`rm datei`	Löschen einer Datei.
`rmdir verzeichnis`	Löschen eines leeren Verzeichnisses.
`mkdir verzeichnis`	Anlegen eines Verzeichnisses.
`yppasswd`	Ändern des Passworts.
`emacs`	Starten des Editors Emacs.
`g++ datei.cpp`	Übersetzen eines Programms *datei.cpp* in eine ausführbare Datei *a.out*.
`g++ datei.cpp -o datei` oder `make datei`	Übersetzen von *datei.cpp* in eine ausführbare Datei *datei*.
`exit` oder `logout`	Ausloggen und Beenden der Sitzung in einer Shell.

Die wichtigsten Kommandos unter Linux

C++ mit Emacs und g++

Wir empfehlen unter Linux als Editor den Emacs und als Compiler den GNU g++ zu verwenden. Die folgenden Schritte geben einen beispielhaften Durchlauf durch eine Programmiersitzung mit dem Emacs und g++ an.

1. Wechseln Sie mit dem *Pager* auf einen freien virtuellen Bildschirm.
2. Starten Sie den Emacs (z.B. mit der linken Maustaste in dem erscheinenden Arbeitsmenu oder über den Button “Tools”).
3. Öffnen Sie mit `Strg-x Strg-f` eine Datei `hello.cc`. Da die Datei noch nicht existiert, wird sie neu erzeugt. (Hinweis: auf die Dauer ist es sinnvoll, mehrere Verzeichnisse für die Aufgaben anzulegen.)
4. Jetzt können Sie einen beliebigen Text eingeben, z.B. folgendes Programm, das ’`Hello, World`’ ausgibt.


```
#include <iostream.h>
int main()
{
   cout << ”Hello, World!” << endl;
   return 0;
}
```

Tip: Benutzen Sie `Tab`, um die Quelltextzeilen einzurücken.

5. Mit `Strg-x Strg-s` wird das eingegebene Programm gespeichert.
6. Das Programm soll nun übersetzt und gestartet werden. Zur Übung wollen wir das zunächst von Hand in der Shell durchführen. Dazu
   (a) Mit dem *Pager* in den virtuellen Bildschirm mit der Shell wechseln oder im Arbeitsmenu eine neue Shell öffnen.
   (b) In der Shell `g++ hello.cc` eingeben. Dadurch wird der eingegebene Quelltext in ein ausführbares Programm (Defaultname `a.out`) übersetzt. Sollten beim Übersetzen Fehler auftreten, so müssen Sie den Programmtext nochmal gründlich untersuchen und diese Fehler beseitigen. Danach muß erneut gespeichert und übersetzt werden.
   (c) War das Übersetzen erfolgreich (der Compiler hat keine Fehlermeldung ausgegeben), so kann das Programm jetzt mit ’`a.out`’ gestartet werden und in der Shell erscheint die Ausgabe

```
Hello, World!
```

`C-x C-f`	Laden oder Erzeugen einer Datei.
`C-x C-s`	Speichern einer Datei.
`C-g`	Abbrechen einer Operation (zurück in den Normalzustand :–).
`C-x C-c`	Beenden des Emacs.
`M-x` *befehl*	Eingabe eines Kommandonamens von Hand.
`C-h f`	Hilfe zu einer Funktion.
`C-h k`	Hilfe zu einer Tastenkombination.
`C-h i`	Startet info. Hier gibt es Hilfetexte zum Emacs, dem gcc, der C++–Standardbibliothek usw.
`C-space`	Start eines Blocks markieren.
`C-w`	Block löschen.
`C-w`	Block an Cursorposition einfügen.
`C-k`	Zeile von Cursorposition bis Ende löschen.
`C-s`	Suchen nach einem Text.
`M-%`	Suchen und Ersetzen.

Die wichtigsten Kommandos im Emacs
Gemäß den Emacs–Konventionen steht '`C-`' für '`Strg-`' und '`M-`' für 'Meta', auf der Tastatur als '`Alt-`' oder '`Esc`'.

Weitere Hinweise

Debugging

Der Emacs kann auch den `gdb`, den Gnu DeBugger, integriert betreiben. Mit `M-x gdb` wird der gdb–Modus aufgerufen, wozu die Kommandozeile abgefragt wird, etwa `gdb Hello`. Es öffnet sich ein Emacs–Fenster mit der gdb–Konsole. `help` zeigt eine Liste der Befehle, hier interessant sind `b` *`funktion`* (Programm bei Eintritt in *`funktion`* unterbrechen), `n` (Programm bis zur nächsten Zeile ausführen), `s` (Einzelschritt, auch in Funktionen hinein), `c` (Programm weiter ausführen), `p` *`ausdruck`* (*`ausdruck`* auswerten und ausgeben). Der Emacs zeigt dabei nach Möglichkeit die passenden Quelltextzeilen an.

Make

Mit `make` kann die Programmerstellung automatisiert werden. Anleitungen hierzu werden Dateien namens `makefile` oder `Makefile`

entnommen. Vieles 'weiß' `make` sowieso, aber ein Makefile mit dem Inhalt `CPPFLAGS=-ansi -pedantic -Wall -g` ist hier nützlich, weil es `make` dazu anweist, beim Compilieren von C++–Quelltexten vom Compiler maximal viele Warnungen zu melden und Informationen für den Debugger abzulegen.

EMail

Post läßt sich im CIP–Pool noch am ehesten mit Netscape (über den Button 'Tools' zu haben) verschicken und lesen. Der zu nutzende Dienst ist POP3, der Benutzername auf dem Server sollte bereits eingetragen sein und das Paßwort ist die Benutzernummer.

Übungsaufgaben in C++

Hier ein paar Aufgaben, die Sie bearbeiten können, um mit dem System vertraut zu werden.

1. Schreiben Sie ein Programm, das nach den folgenden Daten eines Studenten bzw. einer Studentin fragt, sie einliest und anschließend wieder ausgibt: (Name, Vorname, Matrikelnummer und Geschlecht). Sie können folgendes Programmskelett verwenden

```
#include <iostream.h>
char name[30]; // name speichert 29 Zeichen
               // +abschliessendes Nullbyte
// ...
int main()
{
    // Daten eingeben
    cout << "Name eingeben: ";
    cin >> name;
    // ...
    // Daten ausgeben
    cout << "Name: " << name << endl;
    // ...
    return 0;
}
```

2. Die Fibonacci–Zahlen sind wie folgt definiert:
   $Fib(1) = 1$
   $Fib(2) = 1$
   $\forall_{(i>2)}\ Fib(i) = Fib(i-1) + Fib(i-2)$

(a) Schreiben Sie ein Programm, daß eine Zahl `N` einliest und die `N`te Fibonacci–Zahl berechnet und ausgibt.

(b) Erweitern Sie das Programm so, daß die ersten `N` Fibonacci–Zahlen berechnet und ausgegeben werden.

3. Schreiben Sie ein Programm, das zwei positive ganze Zahlen einliest und deren größten gemeinsamen Teiler und kleinstes gemeinsames Vielfaches berechnet und ausgibt.

4. Schreiben Sie ein Programm, daß die Werte der Sinusfunktion im Intervall [0, 2π[berechnet. Das Programm soll die Werte in einem Abstand von $\frac{\pi}{25}$ berechnen, also $\sin 0, \sin\frac{\pi}{25}, \sin\frac{2\pi}{25}, \ldots, \sin\frac{49\pi}{25}$.
Um die Funktion am Bildschirm anzuzeigen, speichern sie die berechneten Werte in einer zweidimensionale Feldvariable mit-Komponenten vom Typ CHAR wie folgt ab: Eine Dimension des Feldes entspricht der X–Achse, die andere der Y–Achse. Das gesamte Feld wird mit Leerzeichen initialisiert. Nun soll für jeden berechneten Funktionswert ein Stern in die Komponente des Feldes geschrieben werden, die der (X,Y)–Position des berechneten Wertes am nächsten kommt. Danach wird das Feld Zeichenweise auf dem Bildschirm ausgegeben (am Ende einer Zeile ein `cout << endl;` (Zeilenumbruch) einfügen).
Ein recht guter Wert für die Größe der Y–Dimension des Feldes ist 24, da das ungefähr der Bildschirmgröße entspricht.
Um einen ganzzahligen Wert `i` in einen reellwertigen umzuwandeln können Sie z.B. `float f = float(i)` verwenden. Umgekehrt können Sie mit `i = int(f)` einen `float` in einen `int` wandeln.

Experimentieren Sie ein wenig mit den Parametern und der dargestellten Funktion.

Anhang IV:

Übungsaufgaben zum Selbststudium

Im folgenden sind die Übungsaufgaben der letzten drei Jahre für sie zusammengestellt.

Diese sollen Ihnen die Gelegenheit zum Selbststudium eröffnen, zusätzlich zu den übungen zur Vorlesung, die bewertet und in den Übungsgruppen besprochen werden.

Die Erarbeitung dieser Übungen empfiehlt sich begleitend zur Vorlesung, insbesondere aber zur Klausurvorbereitung. Sie erhalten damit ein Mittel zur Selbstkontrolle:

a) Habe ich den Stoff verstanden?

b) Kann ich ihn anwenden?

c) Kann ich die Aufgaben in der vorgegebenen Zeit der Klausur lösen?

Die Übungen sind thematisch eingeteilt in:

- Syntax (Aufgaben 1 – 4)
- Programmierung in C++ (Aufgaben 5 – 12)
- Sortieren (Aufgaben 13 – 16)
- Testen (Aufgaben 17 – 19)
- Listen (Aufgaben 21 – 22)
- Bäume (Aufgaben 23 – 26)
- Graphen (Aufgaben 27 – 29)
- Module und Datenabstraktion in C++ (Aufgaben 20, 30 – 34)

Aufgaben die mit * gekennzeichnet sind, sind entweder aufwendig oder behandeln weiterführenden Stoff.

Syntax

1. Aufgabe :

Geben Sie zu jeder der folgenden einfachen Sprachen jeweils eine Grammatik an, mit der alle Worte dieser Sprache erzeugt werden können.

(a) Worte beliebiger Länge, die nur aus dem Buchstaben ’a’ bestehen.

(b) Alle symmetrischen Worte aus den Buchstaben ’a’ und ’b’ (also z.B. ’a’, ’aba’, ’bbbb’, ’abaaba’).

(c) Eine beliebige Folge von Namen (Vorname und Nachname) Ihrer Übungsgruppe durch Komma getrennt. Dabei sollen die Vornamen optional sein.

2. Aufgabe :

(a) Entwickeln Sie eine Grammatik in EBNF-Notation für Literaturangaben der Form:

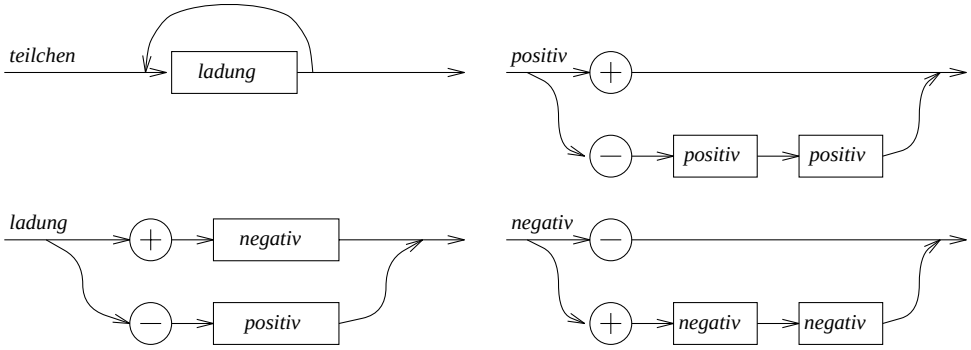
```
Autor/inn/en:  Titel, Verlag, Erscheinungsjahr, Seitenzahl,
referenzierte Seite(n).
```

Mit referenzierte Seite(n) ist/sind die Seiten der Veröffentlichung gemeint, auf die sich die Literaturangabe bezieht. Die Nichtterminale für Wort bzw. Zahl brauchen nicht verfeinert werden!

(b) Übersetzen Sie die EBNF-Regeln aus Teil a) in Syntaxdiagramme. item Können Sie mit Hilfe von EBNF-Regeln oder Syntaxdiagrammen angeben, daß die referenzierte(n) Seite(n) innerhalb der angegebenen Seitenzahl liegen müssen? Wenn ja wie? Wenn nein, warum nicht?

3. Aufgabe :

Gegeben seien die folgenden Synatxdiagramme:



(a) Was für Worte kann man mit diesen Regeln erzeugen (das Startsymbol ist *teilchen*)?

(b) Übersetzen Sie die Diagramme in EBNF-Regeln. item Finden Sie eine Grammatik (in EBNF-Notation oder Syntaxdiagrammen), mit der die gleichen Worte erzeugt werden, die aber weniger Regeln hat.

4. Aufgabe :

Ein Programm in C++ besteht aus unterschiedlichen Komponenten, die die verschiedenen Aspekte eines Programms widerspiegeln. Prinzipiell kann man vier Arten von Bausteinen unterscheiden, aus denen sich ein Programm zusammensetzt.

Grobe Programmstruktur: Hierunter fallen die Aufteilung eines Programms in Prozeduren und Funktionen sowie, insbesondere bei großen Programmen, die Einteilung in Module.

Deklarationen: Alles, was in den Anweisungen eines Programms verwendet wird, muß zuvor deklariert, also bekannt gemacht werden.

Anweisungen: Die Teile eines Programms, die bei seiner Ausführung bestimmen, was geschieht.

Ausdrücke: Will man etwas über die Werte der Variablen eines Programms wissen (z.B. in der Bedingung einer IF-Anweisung) oder deren Werte ändern, so benötigt man Ausdrücke, um mit diesen Werten umzugehen.

In der Syntax einer Programmiersprachen müssen diese Bestandteile der Sprache definiert werden. Betrachten Sie die Grammatik von C++ im Anhang I Skript.

(a) Schreiben Sie für jeden der vier beschriebenen Bereiche 5 Regeln aus der Grammatik auf, die Sie diesem Bereich zuordnen würden.

(b) Können Sie jede Regel eindeutig einem der Bereiche zuordnen? Bei welchen Regeln scheint das nicht möglich zu sein und warum (drei Beispiele mit Begründung).

Der Programmieren-im-Kleinen Teil von C++

5. Aufgabe :

- Welche der folgenden C-Programme sind korrekt und was geben Sie (abhängig vom eingelesenen Wert von a) aus?

1.
```
#include <iostream.h>
void main(void){
char a;
cin >> a;
if (a=='w'){
  cout << "Hallo";
  cout << " Welt!\n";}
else{
  cout << "Hallo ";
  cout << " Erde!\n";}
}
```

2.
```
#include <iostream.h>
void main(void){
char a;
cin >> a;
if (a=='w')
  printf("Hallo");
  cout << " Welt!\n";
else
  cout << "Hallo";
  cout << " Erde!\n";
}
```

3.
```
#include <iostream.h>
void main(void){
char a;
cin >> a;
if (a=='w'){
  cout << "Hallo";
  cout << " Welt!\n";}
else
  cout << "Hallo";
  cout << " Erde!\n";
}
```

4.
```
#include <iostream.h>
void main(void){
char a;
cin >> a;
cout << "Hallo";
if (a=='w')
  cout << " Welt!\n";
else
  cout << " Erde!\n";
}
```

5.
```
#include <iostream.h>
void main(void){
char a;
cin >> a;
cout << "Hallo";
switch(a){
  case 'w':
    cout << " Welt!\n";
  default:
    cout << " Erde!\n";
  }
}
```

6.
```
#include <iostream.h>
void main(void){
char a;
cin >> a;
cout << "Hallo";
switch(a){
  case 'w':
    cout << " Welt!\n";
    break;
  default:
    cout << " Erde!\n";
  }
}
```

- Was geben die folgenden Anweisungen aus (i sei als int i deklariert)?

1. for(i = 0; i < 5; i++) cout « i « " ";
2. for(i = 0; i < 5; i++) {cout « i « " ";}
3. for(i = 0; i < 5; i++); cout « i « " ";
4. for(i = 0; i < 5;) cout « i++ « " ";
5. for(i = 0; i < 5;) cout « ++i « " ";
6. for(i = 0; i < 5;) cout « (++i -1) « " ";

6. Aufgabe :

- In C(++) gibt es die Möglichkeit auch komplexe Datenstrukturen mit Hilfe von Literalen zu initialisieren. Deklarieren Sie ein Dreidimensionales Feld von ganzen Zahlen wuerfel, wobei jede Dimension die Größe 4 hat (also ein Würfel mit Kantenlänge 4). Initialisieren Sie dieses Feld mit beliebigen Werten.

- Für Zeichenketten gibt es eine Schreiberleichterung. Folgende Deklaration ist gültig:

  `char name[80] = "Arthur".`

  Deklarieren Sie ein Zweidimensionales Feld von Zeichen (5×80) und initialisieren Sie es mit verschiedenen Namen.

- Definieren Sie einen Verbundtyp Person, der Komponenten für Namen, Vornamen und Alter einer Person vorsieht. Deklarieren Sie eine Variable dieses Typs und weisen Sie ihr einen Wert zu.

7. Aufgabe :

Schreiben Sie ein Programm in C++, das den Benutzer nach einem Datum fragt, es einliest und berechnet, wieviele Tage es noch bis zum nächsten Weihnachtsfest sind. Realisieren Sie dazu eine Funktion DatumNachJahrestag, die ein übergebenes Datum in die Anzahl der im Jahr bisher vergangenen Tage umrechnet. Die Funktion und das Hauptprogramm sollen in getrennten Dateien gespeichert werden, das Hauptprogramm muß also mit Hilfe einer Header-Datei diese Funktion importieren (Funktionsmodul).

8. Aufgabe :

In dieser Aufgabe sollen Sie einen Datentyp entwerfen, mit dem Sie Ihren Wochenstundenplan speichern können. Dazu benötigen Sie verschiedene Hilfsmittel:

(a) Deklarieren Sie einen geeigneten Aufzählungstyp Wochentag.

(b) Deklarieren Sie einen Verbundtyp Veranstaltung, der den Inhalt und die Art einer Veranstaltung aufnehmen kann. Ein solcher Verbund soll den Titel der Veranstaltung, den Namen des Dozenten, die Art der Veranstaltung (Vorlesung oder Übung) und den Veranstaltungsort aufnehmen können.

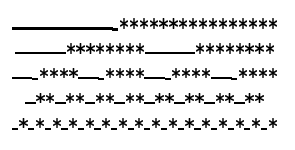
(c) Deklarieren Sie einen zweidimensionalen Feldtyp Stundenplan, dessen erste Dimension den Wochentag und dessen zweite Dimension die Uhrzeit einer Veranstaltung darstellt. Die Komponenten des Feldtyps haben den Typ Veranstaltung. Zur Vereinfachung können Sie davon ausgehen, daß Veranstaltungen nur zur vollen Stunde beginnen und eine Stunde lang dauern.

(d) Weisen Sie einer Variable vom Typ Stundenplan die Werte für Ihre Kleingruppenübung zu (mit gerundeten Anfangs- und Endzeiten).

(e) Wie können die Vereinfachungen, die in Teil c) getroffen wurde aufgehoben werden? Bedenken Sie, daß Veranstaltungen nicht nur jeweils viertelstündlich beginnen können sondern auch unterschiedliche Längen haben. Wie sieht die Zuweisung aus Teil d) jetzt aus?

9. Aufgabe *(Rechneraufgabe)*:

Die vereinigten Staaten von Alphanumerica haben im Zuge der Automatisierung ihrer Flaggenindustrie einen Wettbewerb für die eleganteste Programmierung ihrer Flagge ausgeschrieben. Die Flagge hat folgendes Aussehen:



1. Zeile: k Minus- gefolgt von k Sternsymbolen
2. Zeile: zweimal (k / 2 Minus, k / 2, Sterne)
3. Zeile: viermal (k / 4 Minus, k / 4, Sterne)
4. Zeile: achtmal (k / 8 Minus, k / 8, Sterne)

usw.

Die Flagge gibt es in verschiedenen Größen für k$\in$\{2, 4, 8, 16, 32\} (im Beispiel ist k = 16).

(a) Schreiben Sie ein C++-Programm, das für eine eingelesene Zahl k aus obiger Menge die entsprechende Flagge ausgibt. Das Programm sollte eine **rekursive** Prozedur SchreibeFlagge verwenden, die das Zeichnen einer Flagge übernimmt. Wie sieht die Schnittstelle der Prozedur aus?

(b) Ändern Sie das Programm so ab, daß nun eine nicht-rekursive Prozedur die Flagge ausgibt. Inwiefern abstrahiert die Verwendung einer Prozedur beim Zeichnen der Flagge von der gewählten Lösung?

10. Aufgabe *(Rechneraufgabe)*:

Eine natürliche Zahl $n \in \mathbb{N}$ heißt Primzahl, falls sie nur durch 1 und sich selbst ohne Rest teilbar ist. Ist n keine Primzahl, so gibt es eine Zahl $k \in \mathbb{N}$ mit $k^2 \leq n$, durch die n ohne Rest teilbar ist.

Schreiben Sie ein Programm, das alle Primzahlen ausgibt, die kleiner einer einzulesenden Zahl $m \in \mathbb{N}$ sind. Gestalten Sie das Programm so, daß es leicht lesbar und verständlich ist. Also: Kommentieren Sie ausführlich, und setzen Sie die einzelnen Programmteile textuell voneinander ab (Einrücken).

Hinweis: Zur Lösung können Sie das „Sieb des Erathostenes“ verwenden:

- Schreiben Sie alle Zahlen kleiner m auf, und streichen Sie die 1;
- Nehmen Sie die kleinste noch nicht gestrichene Zahl k, und streichen Sie alle Vielfachen davon (da sie durch k teilbar sind, können sie keine Primzahlen sein);
- Wiederholen Sie den letzten Schritt bis $k^2 > m$ gilt.

11. Aufgabe *(Rechneraufgabe)*:

In dieser Aufgabe sollen zwei Programme entwickelt werden, die das arithmetische Mittel einer Folge von reellen Zahlen in unterschiedlicher Weise berechnen. Die erste Programmvariante speichert die gelesene Zahlenfolge und berechnet nach Eingabe der letzten Zahl einmal das arithmetische Mittel und gibt das Ergebnis aus. Die Länge der Zahlenfolge soll beliebig sein, **die Speicherung der Folge in einer Array-Variablen scheidet also für die Implementierung aus**. Stattdessen soll eine dynamische Datenstruktur (linear verkettete Liste) verwendet werden.

In der zweiten Variante werden die Zahlen nicht gespeichert. Vielmehr merkt sich das Programm nur ein Zwischenergebnis und berechnet nach Eingabe jeder neuen Zahl den Mittelwert der erweiterten Folge (dieser Wert kann dann auch nach jeder Eingabe ausgegeben werden).

Zur Lösung dieser Aufgabe sollten Sie folgende Punkte beachten:

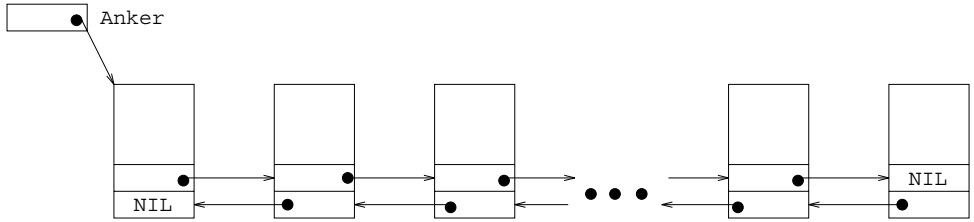
- Überlegen Sie sich einen geeigneten Dialog zwischen Programm und Benutzer. (Insbesondere: wie signalisiert der Benutzer das Ende der Zahlenfolge?)
- Entwerfen Sie einen Datentyp, mit dem die verkettete Liste der ersten Programmvariante realisiert werden kann.
- Wie kann das Zwischenergebnis der zweiten Variante am günstigsten gespeichert werden? Da das Rechnen mit Fließkommazahlen (z.B. den float,double-Zahlen aus C++) immer Ungenauigkeiten mit sich bringt, sollten so wenig Rechenschritte wie möglich gemacht werden.

Welche Variante würden Sie für diese Aufgabe bevorzugen und warum? Welche Variante ist günstiger, wenn z.B. zusätzlich eine grafische Aufbereitung der Zahlenfolge (Meßwerte darstellen) erfolgen soll?

12. Aufgabe *(Rechneraufgabe)*:

In manchen Anwendungen ist es wünschenswert, jeden Eintrag in einer Liste nicht nur mit dem nachfolgenden

Element zu verketten (einfach verkettete Liste), sondern ebenfalls mit dem vorangehenden Element. Dadurch wird beispielsweise das Löschen eines Elements aus der Liste wesentlich vereinfacht. Man hat dann eine *doppelt verkettete Liste*.



Schreiben Sie ein C++ Programm, das eine doppelt verkettete Liste realisiert. Es genügt, wenn sie außer dem Datentyp die Operationen Einfuegen und Streichen implementieren (Einfuegen: z.B. Einfügen vor einem bestimmten Element in der Liste; Streichen: beliebiges Element aus der Liste entfernen). Überlegen Sie sich, wie die Operationen im allgemeinen Fall funktionieren und welche Sonderfälle zu beachten sind.

Sortieren

13. Aufgabe *(Rechneraufgabe)*:

Schreiben Sie eine Prozedur, die eine gegebene doppelt verkettete Liste von Integer/int-Zahlen in der folgenden Weise sortiert: Die Liste wird von Anfang in Richtung Ende einmal durchlaufen und dabei wird jedes Element, das kleiner als das vorangegangene aktuelle Element ist, an seinen richtigen Platz weiter vorne in der Liste einsortiert. Dieses Verfahren wird „Sortieren durch Einfügen“ oder „Straight Insertion“ genannt.

Beispiel: Anfang
12, 3, 7, 19, 10.

1. Schritt: 3 ist das erste Element, das kleiner als sein Vorgänger in der Liste ist, also muß es nach vorne verschoben werden; als Ergebnis erhält man:
3, 12, 7, 19, 10.
Hierbei merkt man sich das nächste noch nicht geprüfte Element, also die 7.

2. Schritt: 7 ist wieder kleiner als 12, an der richtigen Stelle einfügen ergibt dann
3, 7, 12, 19, 10.

3. Schritt: Da 19 korrekt steht, muß nur noch 10 umsortiert werden:
3, 7, 10, 12, 19.

14. Aufgabe :

- Schreiben Sie eine Prozedur *Mergesort*, die eine (einfach oder doppelt) verkette Liste von int-Zahlen in der folgenden Weise sortiert: Zunächst wird die Liste in zwei möglichst gleich große Listen geteilt. Jede der beiden Listen wird nun für sich alleine durch *Mergesort* sortiert. Die beiden sortierten Listen werden nun wieder nach und nach zu einer einzigen zusammengefügt (*merge*) indem man ihre jeweils ersten Elemente miteinander vergleicht und das kleinere in die zusammengesetzte Liste einfügt.

- Schreiben Sie zum Testen der Prozedur ein Programm, das eine Folge von Zahlen einliest, mit der Prozedur Mergesort sortiert und das Ergebnis schließlich wieder ausgibt.

  Skizzieren Sie die Vorgänge beim „Sortieren durch Mischen“ (*Mergesort*) anhand der folgenden Zahlenfolge, indem Sie die internen Zwischenzustände der Zahlenfolge in geeigneter Form darstellen:

  15, 22, 7, 14, 48, 35, 20, 29, 9, 18, 38, 51, 27, 11, 31.

15. Aufgabe $\star$:

Ein elegantes und auch effizientes Sortierverfahren ist Heapsort. Heapsort arbeitet nach dem Prinzip Sortieren durch Auswahl. Der Unterschied zu „Straight Selection“ besteht darin, daß in dem Teil des Feldes, der noch unsortiert ist, eine Struktur aufgebaut wird, die das Auffinden des größten noch nicht sortierten Elementes erheblich beschleunigt.

Ein Heap (Haufen) hat eine baumartige Struktur. Ein Element in einem Heap hat i.a. zwei Nachfolger. Sei A ein Feld mit N Elementen. Die Nachfolger des Elementes mit Index $1 \leq i \leq N/2$ sind $A[2\star i]$ und $A[2\star i + 1]$ (falls $i = N/2$ und N gerade ist, dann hat $A[i]$ nur einen Nachfolger). Alle Elemente mit Index $i > N/2$ haben keine Nachfolger. Sie werden die Blätter des Baumes genannt. Das Element mit Index 1 nennen wir Wurzel. Ein Beispiel für die Abbildung eines Feldes in solch eine Baumstruktur zeigt Abb. 1. Ein Baum, wie er in der Abbildung zu sehen ist, ist aber nur ein Modell, mit dem man sich die Struktur des Heaps besser veranschaulichen kann. Gespeichert wird er in einer Feldvariablen!

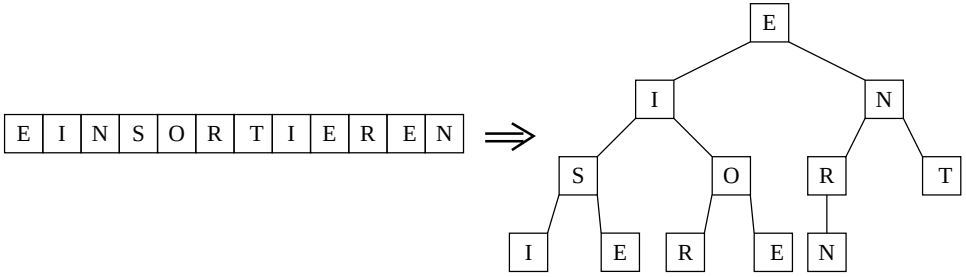


Abbildung 1: Baumartige Anordnung eines Beispielfeldes.

Das dargestellte Feld ist allerdings noch kein Heap. In einem Heap ist jedes Element des Feldes größer als seine Nachfolger. Man kann ein Feld durch Vertauschoperationen in einen Heap überführen. Für das obige Beispiel sieht das Feld bzw. der Baum nach diesen Vertauschungen wie in Abb. 2 aus.

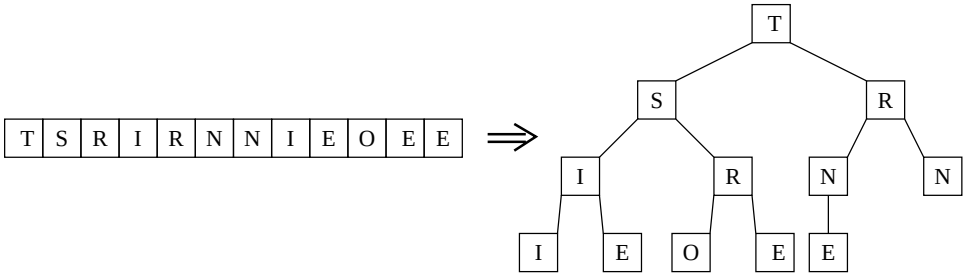


Abbildung 2: Zum Heap umsortiertes Beispielfeld.

(a) Um ein Feld in einen Heap zu überführen, kann man folgerndermaßen vorgehen. Die Elemente mit den Indizes N / 2 + 1 bis N haben keine Nachfolger. Sie sind also bereits Heaps. Nun geht man rückwärts von N / 2 bis 1 und betrachtet die entsprechenden Elemente als Wurzeln von Unterbäumen. In diesen Bäumen muß nur noch untersucht werden, ob die Wurzel an der richtigen Stelle im Baum ist, damit dieser (Unter-) Baum einen (Unter-) Heap bildet. Ist also einer der Nachfolger des untersuchten Feldelementes (der Wurzel des Unterbaumes) größer als die Wurzel, so vertauscht man die Wurzel mit dem größeren der beiden Nachfolger. Dieses Vertauschen muß solange fortgesetzt werden bis das ursprüngliche Wurzelelement nur Nachfolger hat, die kleiner oder gleich groß sind, wie es selbst. Schreiben Sie eine Prozedur HeapAbwaerts, die diese Vertauschungen für einen Unterbaum (gekennzeichnet durch einen Feldindex) vornimmt.

(b) Das eigentliche Sortieren ist nun sehr einfach. Durch die Heapstruktur ist sichergestell, daß das Element mit Index 1 (die Wurzel des Baumes) das größte Element des Feldes ist. Man vertauscht die Wurzel mit dem letzten Element des Feldes, das noch zum Heap gehört. Danach ist die Größe des Heaps um 1 verringert und die Heapeigenschaft muß wiederhergestellt werden, weil das neue Wurzelelement i.a. kleiner als seine Nachfolger sein wird. Diese Schritte werden solange wiederholt, bis der verbleibende Heap die Größe 1 hat. Implementieren Sie dieses Sortierverfahren (dazu gehört auch, vor dem eigentlichen Sortieren, das Feld in einen Heap zu überführen).

(c) Schätzen Sie ab, wieviele Vergleiche im Mittel beim Sortieren mit Heapsort durchgeführt werden müssen. Beachten Sie, daß auch das erstmalige Überführen des Feldes in einen Heap berücksichtigt werden muß.

16. Aufgabe :

Eine Anwendung der Sortierung von Feldern sind Suchalgorithmen. Um einen bestimmten Schlüssel in einem unsortierten Feld der Länge N zu finden, benötigt man im Mittel $\frac{N}{2}$ Vergleiche. Wesentlich bessere Ergebnisse kann man auf sortierten Feldern erzielen. Eines der effizientesten Verfahren ist die binäre Suche.

Gegeben sei ein Feld A. In A sucht man zwischen den Indizes l und h ($l < h$) den Schlüssel s. Dazu betrachtet man zuerst das Element mit dem Index $i = (l + h) / 2$. Gilt A[i] = s, so endet die Suche. Ist A[i] > s, so wiederholt man die Suche in den Indexgrenzen l bis i −1, sonst mit i +1 bis h.

(a) Gegeben sei ein Feld A mit 100 Elementen vom Typ int, mit A[i] = i für alle i von 1 bis 100. Welche vergleiche müssen bei der binären Suche nach dem Schlüssel 35 durchgeführt werden?

(b) Implementieren Sie die binäre Suche auf Feldern mit ganzzahligen Komponenten. Achten Sie besonders auf Terminationsbedingungen und Indexgrenzen (was passiert, wenn der gesuchte Schlüssel nicht im Feld ist).

(c) Zu jedem Programm gehört eine Dokumentation. Während die Verwendung von Kommentaren inzwischen selbstverständlich sein sollte (auch für das Programm aus Teil a), ist weitergehende Dokumentation bei kleinen Programmen eher selten zu finden. Gerade für kompakte Algorithmen erleichtert die zusätzliche Dokumentation aber das Verständnis. Zeichnen Sie daher zu dem in a) entwickelten Programmteil, der die binäre Suche realisiert, ein Struktogramm.

(d) Schätzen Sie ab, wieviele Vergleich im schlechtesten Fall nötig sind, damit die binäre Suche (erfolgreich oder nicht erfolgreich) terminiert. Machen Sie sich den Unterschied zur Suche in einem unsortierten Feld anhand eines Zahlenbeispiels (10000 Elemente in einem Feld) klar.

Testen

17. Aufgabe :

Gegeben sei ein Programm, das drei ganze Zahlen einliest und sie als Längen der Seiten eines Dreiecks interpretiert. Das Programm druckt eine Meldung mit der Feststellung aus, ob das Dreieck ungleichseitig, gleichschenklig oder gleichseitig ist.

Zu diesem Programm sollen Sie einen Blackbox-Test durchführen. Denken Sie sich dazu sinnvolle Testfälle aus und notieren Sie mit Kommentaren, wozu sie dienen sollen.

18. Aufgabe :

Folgender Algorithmus sortiert ein Feld a der Länge n in aufsteigender Reihenfolge mittels „Sortieren durch Einfügen“. Stellen Sie zur Durchführung eines Whitebox-Tests den Flußgraphen für den Algorithmus auf, und geben Sie geeignete Testfälle an, die (zusammengenommen) alle Pfade des Flußgraphen durchlaufen.

```
for (int i=1; i<n; i++) {
   x = a[i]; L = 0; R = i;

   while (L<R) {
      m = (L+R)/2;
      if (a[m] <= x) {
         L = m+1; };
      else {
         R = m;
      };
   };

   // Einfuegestelle (R) gefunden
   for (int j=i; j<=R+1; j--) {
      a[j] = a[j-1];
   };
   a[R] = x;
};
```

19. Aufgabe :

Stellen Sie zur Durchführung eines Whitebox-Tests den Flußgraphen für den in Aufgabe 21 implementierten Algorithmus der binären Suche auf, und geben Sie geeignete Testfälle an, die (zusammengenommen) alle Pfade des Flußgraphen durchlaufen.

Module und Datenabstraktion

20. Aufgabe :

In der Vorlesung sind die Schnittstelle und der Rumpf eines abstrakten Datenobjekts „Buffer“ (Synonyme: Puffer, Schlange, Queue) vorgestellt worden. Ein Puffer ist eine sequentielle Datenstruktur, bei der Elemente an einem Ende eingefügt und am anderen Ende ausgelesen werden (FIFO: first in first out). Bei einem „Keller“ (Synonyme: Stapel, Stack) hingegen werden die Datenelemente am gleichen Ende eingefügt und wieder entnommen (LIFO: last in first out).

- Wie müßte die Schnittstelle eines abstrakten Datenobjekts „Keller“ aussehen? Definieren Sie diese Schnittstelle in C++ und implementieren Sie den Rumpf dazu.

- Definieren Sie die Schnittstelle eines abstrakten Datentyps „Keller“. Was ist anders als beim abstrakten Datenobjekt.

Datenstruktur Liste

21. Aufgabe :

In der Vorlesung haben Sie als Realisierungstechniken für Listen unter anderem die *nicht zirkuläre sequentielle Realisierung*, die *einfach verkettete Zeigerrealisierung* und die *einfach verkettete Indexrealisierung* kennengelernt.

(a) Schreiben Sie für jede der genannten Techniken eine Prozedur, die ein Element nach dem aktuellen Element in eine übergebene Liste einfügt (Typdefinitionen wie im Skript):

```
void ElementEinfuegen(Datenstr Liste, text Element);
```

(b) Diskutieren Sie die Vor- und Nachteile der genannten Realisierungstechniken.

22. Aufgabe ⋆:

In dieser Aufgabe sollen Sie einen abstrakten Datentypmodul Liste implementieren, der eine lineare Liste verkapselt. Die Liste soll Elemente vom Typ ElementTyp (von Ihnen zu wählen) speichern. Der Modul soll folgende Operationen zur Verfügung stellen:

- Erzeugen und Löschen von Listen,
- Abfragen, ob eine Liste leer oder voll ist,
- Einfügen von Elementen in eine Liste, Löschen von Elementen aus einer Liste (am Ende, am Anfang der Liste, vor oder nach dem aktuellen Element (s.u.))
- Abfragen, ob ein Element in einer Liste vorhanden ist

Auf die Elemente der Listen soll nur über ihren Wert zugegriffen werden können (es werden also keine Zeiger / Indizes von Elementen über die Schnittstelle gereicht). Um trotzdem Elemente der Listen an bestimmten Positionen löschen / einfügen zu können, soll zu jeder Liste ein „aktuelles“ Element existieren, auf das sich Operationen wie Löschen und Einfügen beziehen. Dazu benötigt man weitere Operationen:

- Gehe zum Anfang / Ende einer Liste
- Gehe zum nächsten / vorherigen Element der Liste

(a) Schreiben Sie die Schnittstelle des Moduls in C++. Kommentieren Sie die Schnittstellenoperationen ausführlich.

(b) Implementieren Sie den Rumpf des Moduls. Dabei können Sie zwischen verschiedenen Realisierungen von Listen wählen: Felder, einfach oder doppelt verkettete Listen. Achten Sie darauf, daß zu jeder Liste neben der reinen Listeninformation auch noch das aktuelle Element verwaltet werden muß.

(c) Implementieren Sie einen Modul (das Hauptprogramm), daß den Modul Liste benutzt. Das Programm soll eine kleine Liste aufbauen, die dann mit dem Verfahren „Straight Selection“ sortiert wird.

(d) Was ändert sich an der Schnittstelle des Moduls, wenn statt eines abstrakten Datentyps ein abstraktes Datenobjekt realisiert werden soll? Was muss geändert werden, wenn ein anderer Datentyp als der von Ihnen gewählte gespeichert werden soll?

Datenstruktur Baum

23. Aufgabe *Allgemeiner Baum*:

(a) Berechnen Sie zum Binärbaum aus Abbildung 3 Preorder-, Postorder und Inorder-Durchläufe.

(b) Gegeben Sei die nebenstehende Schnittstelle eines ADT Baum. Schreiben Sie eine rekursive C++-Prozedur Postorder, die durch einen Baum (vom Typ Tree) einen Postorder-Durchlauf macht und die Markierungen (Label) seiner Knoten dabei ausgibt.

(c) Bei Binärbäumen mit eindeutigen Markierungen ist es möglich aus gegebenen Preorder- und Postorder-Durchläufen den ursprünglichen Baum zu rekonstruieren. Geben Sie einen Algorithmus an (informell), der dies leistet und wenden Sie ihn auf das folgende Beispiel an.

Preorder:	10	5	3	16	12	20	18	23
Postorder:	3	5	12	18	23	20	16	10

Die Rekonstruktion ist nicht immer eindeutig möglich. Wann gibt es bei der Rekonstruktion Wahlmöglichkeiten und welche Information braucht man, um die Rekonstruktion eindeutig zu machen?

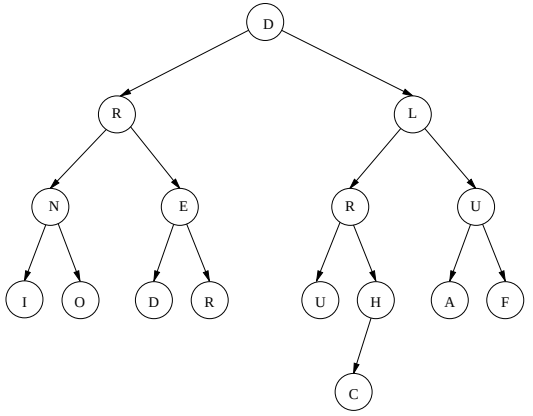


Abbildung 3: Binärbaum

24. Aufgabe *Balancierte Bäume*:

Gegeben sei folgende Typdeklaration zum Aufbau der Teile eines Binärbaums:

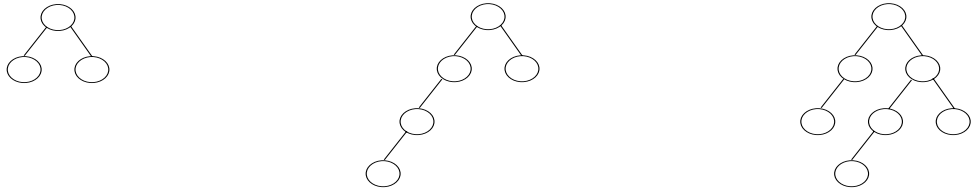
```
class BinBaum {
```

```
class ADTTree {
public:
   typedef unsigned int NodeIdT; // Jeder Knoten besitzt eine eindeutige Nummer

   ADTTree();

   void makeEmpty();
   bool isEmpty();

   NodeIdT root();

   NodeIdT leftmostChild(NodeIdT node);
   NodeIdT rightSibling(NodeIdT node);
   unsigned int label(NodeIdT node);
};
```

Abbildung 4: Schnittstelle eines ADT für Bäume

```
   unsigned int inhalt;
   BinBaum * links;
   BinBaum * rechts;
};
```

Eine Definition für die Höhe eines Baumes lautet: Die Höhe eines Baumes, der nur einen Knoten hat ist 1. Die Höhe eines beliebigen Baumes ist das Maximum der Höhen seiner Teilbäume plus eins. Ein Binärbaum heißt balanciert, wenn für jeden Knoten gilt, daß die Höhe des linken Teilbaums sich von der Höhe des rechten Teilbaums höchstens um den Wert eins unterscheidet. Hierzu einige Beispiele:



balanciert
Hoehe = 2

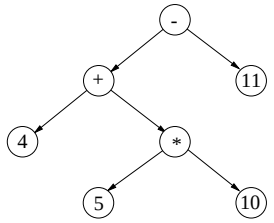
nicht balanciert
Hoehe = 4

balanciert
Hoehe = 4

Schreiben Sie eine rekursive C++-Prozedur istBalanciert, die ermittelt, ob ein gegebener Binärbaum balanciert ist oder nicht.

25. Aufgabe :

Binäre Bäume eignen sich gut, um arithmetische Ausdrücke zu speichern. Die Blätter solcher Bäume enthalten die Zahlen (oder Variablen) des Ausdrucks, die inneren Knoten repräsentieren die Operatoren. Ein Beispiel: Der Ausdruck $(4 + 5 * 10) - 11$ ergibt den nachfolgenden Baum.



(a) Entwerfen Sie eine Datenstruktur Ausdruck, die einfache Ausdrücke baumartig speichert. Es soll nur mit Zahlen, nicht mit Variablen gerechnet werden. Als Operationen sind nur Addition, Subtraktion, Multiplikation und Division zugelassen (ganze oder reelle Zahlen).

(b) Implementieren Sie eine Funktion Auswertung, die eine Variable vom Typ Ausdruck als Eingabe erhält, den übergebenen Ausdruck auswertet und das Ergebnis zurückliefert.

(c) Geben Sie eine Grammatik an, mit der die oben angeführte Art von Ausdrücken beschrieben wird (auf Präzedenz achten — Punkt- vor Strichrechnung). Wie unterscheidet sich der Expression-Baum eines Ausdrucks von seinem Ableitungsbaum?

26. Aufgabe :

In der Vorlesung haben Sie verschiedene Realisierungsmöglichkeiten für Bäume mit beliebigem Auslaufgrad (N-äre Bäume) kennengelernt. Wir betrachten hier zwei davon: „Knoten mit festem Auslaufgrad und Überlaufwächter“ und „LeftmostChild-RightSibling“.

(a) Entwerfen Sie MODULA-2 Datentypen, die diesen Realisierungen entsprechen.

(b) Schreiben Sie für beide in a) entworfenen Datentypen eine Prozedur, die ein Kind unter einen Knoten einfügt.

(c) Schreiben Sie für beide in a) entworfenen Datentypen eine Prozedur, die die Höhe eines Baumes ermittelt.

Datenstruktur Graph

27. Aufgabe :

- Realisieren Sie ein Datenobjektmodul UnmarkierterGerichteterGraph mit Hilfe einer Adjazenzmatrixdarstellung. Dabei sei die maximale Anzahl von Knoten im Graphen konstant. Eine Adjazenzmatrix ist eine Matrix $AdjazenzM$ mit Booleschen Einträgen für die gilt:

$$AdjazenzM[i,j] = \begin{cases} TRUE & \text{falls es eine Kante von Knoten i zu Knoten j im Graphen gibt} \\ FALSE & \text{sonst} \end{cases}$$

- Wie müssen Sie Ihre Realisierung ändern / erweitern, wenn Kanten und / oder Knoten im Graphen markiert sein dürfen?

28. Aufgabe :

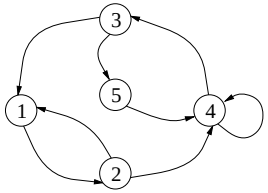
In Aufgabe 39 haben Sie ein Datenobjektmodul UnmarkierterGerichteterGraph erstellt, das einen Graphen mit Hilfe einer Adjazenzmatrix realisierte. Verwenden Sie dieses abstrakte Datenobjekt, um ein Datenobjektmodul UnmarkierterUngerichteterGraph zu entwerfen.

- Unterscheiden sich die Schnittstellen von UnmarkierterGerichteterGraph und UnmarkierterUngerichteterGraph? Begründen Sie.
- Schreiben Sie Prozeduren zum Einfügen und Löschen von Kanten in den ungerichteten Graphen, wie Sie im Rumpf des Datenobjektmoduls stehen könnten. Hat die Implementierung von UnmarkierterGerichteterGraph als Adjazenzmatrix Einfluß auf die Realisierung der Prozeduren?

29. Aufgabe :

In der Vorlesung haben Sie verschiedene Realisierungsmöglichkeiten für die knotenorientierte Speicherung (Adjazenz-Darstellung) von Graphen kennengelernt. Von den vorgestellten Möglichkeiten beinhaltet nur die charakteristische Adjazenzmatrix direkte Informationen über in Knoten einlaufende Kanten. Diese Information ist aber wichtig, wenn man z.B. das Löschen von Knoten effizient implementieren will (warum?).

(a) Entwerfen Sie drei Datenstrukturen, die die sequentiellen Adjazenzlisten, die verdichteten seq. Adjazenzlisten und die verketteten Adjazenzlisten so erweitern, daß zu jedem Knoten Informationen über seine einlaufenden Kanten gespeichert werden. Stellen sie die entworfenen Strukturen skizzenhaft dar (Zeichnungen analog zu denen im Skript). Als Beispiel für die Skizzen soll folgender Graph dienen.



(b) Geben Sie für eine der von Ihnen entworfenen Datenstrukturen eine entsprechende Typdefinition in C++ an.

(c) Wie groß ist der Aufwand zum herausfinden aller einlaufenden Kanten in einen Knoten bei den im Skript vorgestellten Realisierungsmöglichkeiten und wie groß ist er bei Ihren Datenstrukturen (Graph mit n Knoten und m Kanten)? Womit erkaufen Sie sich diesen Vorteil?

(d) Entwerfen Sie eine Datenstruktur, bei der nicht nur die Adjazenzlisten verkettet sind, sondern ebenfalls die Knoten des Graphen in einer verketteten Liste gespeichert werden. Wie sollten dabei sinnvollerweise die Ziel- bzw. Quellknoten in den Adjazenzlisten identifiziert werden? (Zeichnung und Typdefinition)

Module und Datenabstraktion in C++

30. Aufgabe :

Schreiben Sie in C++ die Schnittstelle eines Datenobjektmoduls, das eine Menge von Personendaten verwaltet. Das Modul soll die Operationen Einfuegen, Loeschen und Suchen anbieten.

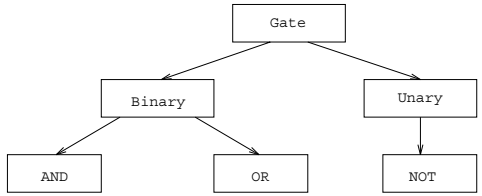
Im Rumpf dieses Moduls sollen die Personendaten in einer doppelt verketteten Liste nach den Nachnamen sortiert gespeichert werden. Um einen effizienteren Zugriff auf die Elemente der Liste zu haben, wird statt eines einzelnen Ankerzeigers ein Feld von 26 Ankern vereinbart, die jeweils auf die erste Person in der Liste, deren Nachname mit einem bestimmten Buchstaben des Alphabetes beginnt, zeigen sollen. Implementieren Sie folgende Teile des Rumpfes: die Deklarationen der benötigten Variablen und die Funktion zum Einfuegen von Personendaten.

31. Aufgabe :

Definieren Sie in C++ eine Klasse BinBaum, die einen abstrakten Datentyp „Binärer Baum“ verkapselt. Die Klasse sollte Methoden zum Einfügen und Löschen von Knoten sowie zum Navigieren durch den Baum besitzen (vgl. Aufgabe 35).

32. Aufgabe :

Im folgenden sollen einfache logische Schaltungen simuliert werden. Dazu werden einige Klassen in C++ definiert, die diese Bauteile nachbilden. Die Basisklasse ist die Klasse Gate. Weitere Schaltelemente sollen gemäß der folgenden Klassenhierarchie ebenfalls vereinbart werden:



Wir unterscheiden also Gatter mit zwei Eingängen und Gatter mit einem Eingang. Alle Gatter haben allerdings nur einen Ausgang. Um aus solchen Objekten kleine Schaltnetze aufbauen zu können, soll jede Klasse die Methoden void setInput(int n, logic value) (setzen des Eingangswertes für Eingang Nr. n) und logic getOutput(void) (auslesen des aktuellen Ausgabewertes) besitzen. Der Typ logic sei dabei ein Aufzählungstyp mit den Werten 0 und 1.

- Implementieren Sie die angegebenen Klassen. Dabei sollen die Methoden setInput und getOutput virtuell definiert werden.
- Definieren Sie eine Klasse Halfadder, deren Objekte sich aus den für einen Halbaddierer benötigten Gatter-Objekten zusammensetzen. Ein Halfadder soll ebenfalls Methoden zum setzen der Eingabewerte und zum Auslesen der Ausgabewerte (allerdings hier für zwei Ausgänge) besitzen. Die Ausgabewerte sollen mit Hilfe der Gatter-Objekte berechnet werden.

33. Aufgabe :

Implementieren Sie in C++ eine Klasse ListenElement und eine Klasse Liste. Objekte vom Typ Liste sollen Objekte vom Typ ListenElement (und davon abgeleitete) in einer doppelt verketteten Liste speichern können. Schreiben Sie für Liste einen Konstruktor, der Objekte als leere Listen initialisiert und einen Destruktor, der alle in der Liste gespeicherten Objekte ebenfalls zerstört.

34. Aufgabe ⋆:

In C++ können Klassen auch von mehr als einer Basisklasse abgeleitet werden (Mehrfachvererbung). Die Definition einer Klasse C, die von zwei Basisklassen A und B erbt, sieht so aus:

```
class C: A, B {...}
```

Betrachten Sie nun die folgenden Fortbewegungsmittel: Amphibienfahrzeug, Cabrio, Fahrzeug, Landfahrzeug, LKW, Motorboot, PKW, Segelboot, Wasserfahrzeug, Yacht.

- Stellen Sie für diese Begriffe eine sinnvolle Klassenhierarchie auf, in der auch Mehrfachvererbung vorkommt.
- Jede Klasse soll eine Methode besitzen, mit der man den jeweiligen Fahrzeugtyp erfragen kann. Definieren Sie diese Methode als virtuelle Methode in der Klasse Fahrzeug und redefinieren Sie sie in einer der Unterklassen.

Anhang V:

Klausuraufgaben

Die bisherigen Klausuraufgaben incl. Musterlösungen wurden der Fachschaft Elektrotechnik zur Verfügung gestellt. Diese übernimmt freundlicherweise die Vervielfältigung.

Wenden Sie sich also bitte an die Fachschaft, wenn sie einen Satz Kopien erhalten wollen.

Anhang VI:

Glossar

Glossar

Ablaufverfolgung

oder Trace wird ein Durchlauf durch ein Programm genannt, wobei jeweils der veränderte Datenspeicher inspiziert wird. Ein Trace kann am Schreibtisch erfolgen (Walkthrough genannt). Im allgemeinen stellt ein Programmiersystem eine Werkzeugunterstützung zur Verfügung.

Abstraktes Datenobjektmodul

ein Modul, der in der Architektur eines Softwaresystems ein abstraktes Datenobjekt in Form eines Datenabstraktions-Bausteins verkapselt.

Abstraktes Datentypmodul

Modul der Architektur, mit dessen Hilfe abstrakte Datenobjekte zur Laufzeit eines Programms erzeugt werden können. In C++ oder Modula-2 erfolgt die Erzeugung über eine Erzeugungsanweisung. Dabei entstehen Verweise auf erzeugte Objekte (abstrakte Datentypmodule mit Zeigersemantik). Diese Handhabung sollte an der Schnittstelle des adt-Moduls nicht sichtbar sein.

Äquivalenz von Programmen

Zwei Programme in einer Programmiersprache heißen stark äquivalent, wenn die Programmiersprachen-Maschine bei beliebiger Programmausführung die gleichen internen Zustände durchläuft. Daraus ergibt sich, daß beide Programme nur triviale Unterschiede besitzen dürfen. Bei schwacher Äquivalenz muß das Ein-/Ausgabeverhalten jeder Berechnung übereinstimmen, intern können beliebige andere Zustände auftreten. Eine effizientere Fassung eines Programms ist zu der ursprünglichen somit schwach äquivalent.

Aggregate

dienen dem direkten Hinschreiben eines konstanten Werts in einem Programm (zusammengesetztes Literal). In C++ gibt es Aggregate nur für Felder. In Modula–2 gibt es keine Aggregate.

Aktualparameter

→ Parameterzuordnung

(Allg.) Benutzbarkeitsbeziehung

Festlegung, daß für die Realisierung eines Bausteins ein anderer Baustein genutzt werden soll (Importe). In Modula-2 werden dabei sogar einzelne Ressourcen dieses Bausteins importiert, in C++ immer der ganze Baustein.

Anweisungen

Anweisungen dienen der Strukturierung der Berechnung (des Anweisungsteils) eines Programmes oder eines Programmbausteins. Durch Benutzung verschiedener Anweisungsformen (→ Kontrollstrukuren) können beliebige Abläufe zur Laufzeit festgelegt werden. Es ist darauf zu achten, daß die statische Festlegung des Anweisungsteils möglichst die Ausführungsreihenfolge wiedergibt (sparsame Verwendung von Sprüngen, übersichtliche Kontrollstrukturen). Anweisungen werden unterschieden in einfache Anweisungen (Wertzuweisung, Prozeduraufruf) und zusammengesetzte Anweisungen (→ Kontrollstrukturen).

Arrays → Felder

Aufzählungsdatentypen

selbstdefinierte oder vordefinierte Datentypen (in C++ `char`, CHAR in Modula-2) in denen sämtliche Werte des Datentyps explizit aufgezählt werden; mit enum in C++.

Ausdrücke

Die Auswertung von Ausdrücken liefert zur Programmlaufzeit Werte. Diese dienen dazu, Variable zu verändern (Zuweisung), Rückgabewerte zu berechnen (Rücksprunganweisung), Eingabewerte für Unterprogramme festzulegen usw. Eine Spezialform von Ausdrücken sind Boolesche Ausdrücke, mit denen ein Teil des Kontrollflusses gesteuert wird. Relationale Ausdrücke sind wiederum ein Spezialfall Boolescher Ausdrücke. Bezüglich der Syntax besitzen Ausdrücke zum einen eine Aufbausyntax (z.B. muß ein binärer Ausdruck zwei Teilausdrücke besitzen). Zum anderen geben kontextsensitive Regeln weitere Restriktionen vor (z.B., daß beide Teilausdrücke vom gleichen Typ sein müssen).

Ausgangs(gabe)parameter

→ Parameterzuordnung

Auswahlanweisung

Sie ist eine Form der Fallunterscheidung (→ Kontrollstrukturen, → Anweisungen) und dient der Unterscheidung verschiedener Fälle, wobei die Auswahl der alternativen Anweisungsfolgen durch Aufzählen aller entsprechenden Werte für eine Anweisungsfolge vorgenommen wird. Diese Werte werden in Auswahllisten angegeben. Werte können einzeln aufgezählt werden sowie durch Unterbereiche angegeben werden. Alle möglichen Werte des Auswahlausdrucks sollten in den Auswahllisten erscheinen.

Bäume

Bäume oder Wurzelbäume sind spezielle Graphen (Datenstrukturen), bei denen jeder Knoten bis auf die Wurzel genau einen Vorgänger besitzt. Wir unterscheiden beliebige n-äre Bäume von binären Bäumen (jeder Knoten hat maximal 2 Nachfolger). Binäre Bäume werden zum Suchen und inkrementellen Sortieren eingesetzt (→ binäre Suchbäume). Zu jedem n-ären Baum gibt es einen binären (sogen. Leftmost–child–right–sibbling–Darstellung). Bäume können knotenorientiert, mit Knotenelementen realisiert werden, die Verweise auf die Nachfolger enthalten. Alternativ dazu können für die Kantendarstellung eigenständige Kantenelemente eingeführt werden (für alle auslaufenden Kanten je ein Kantenelement oder für alle einlaufenden).

bedingte Anweisung

Sie ist eine Form der Fallunterscheidungen (→ Kontrollstrukturen, → Anweisungen) und dient der Programmierung verschiedener Anweisungsfolgen, die bei Zutreffen einer Booleschen Bedingungen ausgeführt werden. Spezialfälle sind einseitig und zweiseitig bedingte Anweisungen. In der allgemein bedingten Anweisung mit `ELSIF` in Modula-2 können beliebig viele Alternativen auftreten. Diese allgemeine bedingte Anweisung muß in C++ durch geschachtelte zweiseitig bedingte Anweisungen simuliert werden.
(`else if`)

bedingte Schleifen

Formen von Schleifen (→ Kontrollstrukturen, → Anweisung), die durch einen Booleschen Ausdruck gesteuert werden. While–Schleifen machen die Prüfung auf wiederholte Ausführung des Schleifenrumpfs am Anfang, until–Schleifen nach Ausführung des Schleifenrumpfes. Für bedingte Schleifen müssen Terminationsüberlegungen angestellt werden, damit ein Programm nicht in eine Endlosschleife gerät. Der Boolesche Ausdruck muß im Schleifenrumpf verändert werden, da sonst eine Endlosschleife oder eine triviale Schleife entsteht (nicht ausgeführt, nur einmal ausgeführt).

binäre Suchbäume

Binäre Suchbäume sind binäre Bäume, bei denen jeder Knoten einen Suchschlüssel enthält. Sie sind so angeordnet, daß der linke Teilbaum eines Knotens nur kleinere und der rechte nur größere Schlüssel als der Knoten enthält. Im Falle eines ausgeglichenen Suchbaums ergibt sich so eine sehr schnelle Suche (O(log n)). Zur Vermeidung von Unbalancierung werden verschiedene Formen ausgeglichener Bäume definiert. Nach jeder Baumänderungsoperation oder nach Auftreten größerer Unbalanciertheit wird der Baum ausgeglichen.

Bit-Operatoren

In C++ für ganzzahlige Datenobjekte definiert. In Modula-2 existieren diese für die Teilmengenhandhabung (→ charakteristische Darstellung) durch Mengenoperatoren, die auf Bit-Vektor-Operationen zurückgespielt werden.

Black−Box−Test

Modultest nur unter Betrachtung der Export-Schnittstelle eines Moduls. Dieser Test berücksichtigt somit nicht die interne Realisierung des Bausteins. Für den Black-Box-Test ist eine saubere Gestaltung der Schnittstelle zwingend erforderlich. Es gibt spezielle Methoden, die die Testfälle für Ein-/Ausgabedaten aus den Wertebereichen ihrer Bestandteile zusammensetzen.

Blockstruktur

Unterprogramme, in manchen Sprachen auch Blöcke, können ineinander geschachtelt werden. Hierbei können lokale Berechnungen eingeführt werden. Die eingeführten Deklarationen haben einen Gültigkeitsbereich (→ Gültigkeitsbereich) und einen Sichtbarkeitsbereich (→ Sichtbarkeitsbereich). Schachtelung gibt es in Modula–2 nur auf der Ebene von Unterprogrammen, in C++ nur mit Blöcken. Die Schachtelung von Modulen in Modula–2 ist wertlos, da sie nicht mit Hilfsmitteln zur getrennten Bearbeitung verbunden ist.

Boolescher Datentyp

Datentyp `bool` (in Erinnerung an den Mathematiker Boole) in C++ für zwei Werte, nämlich wahr (true) oder falsch (false).

Bubblesort

Direktes Sortierverfahren nach der Strategie ”Sortieren durch Vertauschen”.

C++

Erweiterung der Sprache C, die bereits in den 60er Jahren festgelegt wurde (um Objektorientierung). C++ hat eine weite Verbreitung in der industriellen Praxis. C bzw. C++ sind implementierungsnah gestaltet und erlauben damit das Schreiben effizienter Programme.

Call by Reference

(Aufruf über die Adresse) → Parameterzuordnungs-Mechanismus, → Referenzen

Call by Value

(Aufruf über den Wert) → Parameterzuordnungs-Mechanismus

Datenabstraktionsprinzip

Das Datenabstraktionsprinzip beinhaltet (a) Zugriffsoperationen auf logische Sicht nach außen, (b) Verkapselung der Realisierungsdetails von Modulen, die Daten (Zustand, Gedächtnis) handhaben, innerhalb eines Moduls. Das Zerlegen von Softwaresystemen unter Beachtung dieses Prinzips nennt man objektbasierte Softwarekonstruktion oder objektbasiertes Entwerfen. Dieses ist eine Vorstufe der objektorientierten Konstruktion. Die Beachtung des Datenabstraktionsprinzips ist Voraussetzung für änderbare Softwaresysteme (s. Jahr-2000-Problem). Alle komplexen Daten sind in einem Softwaresystem als Bausteine vertreten.

Datentypkonstruktoren

Konstrukte (→ Konstrukte von Programmiersprachen) zur Festlegung der Datenstrukturen eines Programms. Hierzu zählen das Bilden von Feldern, Verbunden, Zeigern, Mengen etc. Datentypkonstruktoren sind ineinander einsetzbar, z.B. Feld von Verbunden, die Komponenten des Verbundes wiederum beliebig strukturiert. Diese Einsetzbarkeit gilt allerdings nicht für alle Datentypkonstruktoren. So darf z.B. in Modula-2 der Datentypkonstruktur `SET OF` nur auf skalare Datentypen (Aufzählungsdatentypen, ganzzahlige Datentypen, Unterbereiche hiervon) angewendet werden.

Deklarationen

Deklarationen dienen der Festlegung zu verwendender Konstanten, Typen, Datenobjekte, Unterprogramme und Module. Deklarationen für Konstanten, Typen und Datenobjekte stehen im Deklarationsteil einer Programmeinheit (Unterprogramm, Modul, Klasse). Deklarationen müssen in Modula-2 vorab stehen, in C++ sollten sie vorab stehen. Die zusätzliche Festlegung durch Deklarationen erlaubt eine Reihe von Überprüfungen zur Compilezeit. Durch Konstantendeklarationen wird ein Bezeichner für einen compilezeitbestimmten Wert eingeführt, eine Typdeklaration führt einen Namen für eine Art von Objekten ein, deren Struktur festgelegt wird, eine Objektdeklaration führt eine Variable eines bestimmten Typs ein, eine Unterprogrammdeklaration führt die Schnittstelle eines Unterprogramms ein.

Direktes Auswählen

Direktes Sortierverfahren nach der Strategie des ”Sortierens durch Auswählen”.
(selectionsort)

Direktes Einfügen

Direktes Sortierverfahren nach der Strategie des ”Sortieren durch Einfügen”.
(insertionsort)

Dynamischer Speicherbereich

setzt sich aus → Laufzeitkeller und → Halde zusammen.

EBNF

erweiterte Backus–Naur–Form (→ Syntaxnotationen) → Anhang I

Eingangs(gabe)parameter

→ Parameterzuordnung

Endlosschleife

Eine Form von Schleifen (→ Kontrollstrukturen, → Anweisungen). Hier muß insbesondere auf Termination geachtet werden. Die Schleife muß durch einen Sprung (break in C++) verlassen werden. Dabei steuert ein Boolescher Ausdruck das Verlassen der Schleife. Dieser Boolesche Ausdruck muß durch den Schleifenrumpf verändert werden.

Feldaggregat

Ein Feldaggregat ist ein zusammengesetztes Literal, mit dem ein Wert eines Feldes im Programm direkt hingeschrieben werden kann (wie ein Literal für einen skalaren Datentyp). Dies vermeidet die Initialisierung von Feldern über Schleifen, wenn die Komponentenwerte statisch sind.

Felder (Reihungen, engl. Arrays)

Zusammensetzung gleichartiger Datenobjekte zu einem Ganzen. Der Zugriff erfolgt zur Laufzeit mit einem Indexausdruck. Man unterscheidet eindimensionale Felder und mehrdimensionale Felder. Ebenso wird zwischen statischen Feldern (Größe zur Programmerstellungszeit bekannt) sowie dynamischen Feldern (Größe laufzeitabhängig) unterschieden.

Feldtypen mit unspezifizierten Grenzen

→ offene Felder

Flußdiagramm

Graphische Darstellung für ein Programm. Dabei wird nur die Ablaufkontrolle dargestellt. Kontrollstrukturen gibt es nur auf primitiver Ebene (bedingte, unbedingte Sprünge sowie Unterprogrammsprung). Die Form der

graphischen Elemente ist genormt (DIN 66001). → Anhang II

Formalparameter

→ Parameterzuordnung

Funktionsmodul

Zusammenfassung verschiedener Funktionen in einem Modul. Diese sollten dabei das gleiche Ein-/Ausgabeprofil besitzen. Funktionsmodule haben Transformations- oder Berechnungscharakter und besitzen im Gegensatz zu Modulen für Datenabstraktion keinen Zustand, der aufgehoben wird.

ganzzahlige Datentypen

`short, int, long` und `unsigned`-Formen in C (`INTEGER, CARDINAL` in Modula–2) zur Deklaration ganzzahliger Datenobjekte (Zahlenwerte) mit den üblichen Operationen.

getrennte unabhängige Übersetzung

Heutige Programmiersprachen und damit auch C++ und Modula-2 gestatten die separate Übersetzung verschiedener Teile eines Programmsystems. Dies erleichtert die arbeitsteilige Softwareentwicklung. Man spricht von unabhängiger Übersetzung, wenn bei separater Übersetzung keine oder nur unzureichende Querprüfungen erfolgen. Bei getrennter Übersetzung folgt die Prüfung genauso streng, als wenn das ganze Programmsystem dem Übersetzer vorgelegen hätte. Modula-2 ist nah an getrennter Übersetzung, C++ bietet einen Zwischenzustand zwischen unabhängiger und getrennter Übersetzung.

Gültigkeitsbereich

Dies ist der statische Bereich des Quelltextes, in dem eine Deklaration verwendet werden kann. Der Gültigkeitsbereich einer Deklaration reicht von der Deklaration selbst bis zum Ende der entsprechenden Programmeinheit (Unterprogramm oder Modul). Außerhalb dieses Bereichs darf keine Verwendung hingeschrieben werden.

Graphen

Allgemeinste Datenstruktur zum Ablegen/ Auffinden beliebiger netzartig strukturierter Sachverhalte. In praxi sind Graphen markiert, d.h. Knoten und Kanten gehören verschiedenen Arten an. Knotenorientierte Realisierungen legen Nachfolger– oder Vorgängerknoten in Listen ab (sequentiell oder verkettet). Charakteristische Speicherung (Adjazenzmatrix) speichert nicht die Kante, sondern nur die Information, ob eine Kante vorliegt. Kantenorientierte Realisierung behandeln Kanten als eigenständige Einheiten.

Halde

mit dem Anlegen von Speicher mittels `new` in C++ bzw. `ALLOCATE` in Modula-2 wird auf einem besonderen Speicher, Halde genannt, ein sogenanntes Haldenobjekt erzeugt. Darüber hinaus wird ein Zeigerwert darauf eingerichtet. Mit `delete` bzw. `DEALLOCATE` erfolgt die Freigabe. Anlegen bzw. Freigeben erfolgt über den Anweisungsteil eines Programms. Weitere Erläuterungen siehe → Zeiger.

Header-File

werden in C++ per Konvention zur Simulation von Modulen als Modulschittstelle eingesetzt (→ Simulation von Modulen).

Heapsort

Verbessertes Sortierverfahren nach der Strategie ”Sortieren durch Auswählen”.

Implementation-File

werden in C++ per Konvention zur Simulation von Modulen als Modulrumpf verwendet (→ Simulation von Modulen).

Information Hiding

ist ein wichtiges Prinzip bei der Softwarekonstruktion. Realisierungsdetails werden verborgen (einer Funktion, eines Bausteins einer bestimmten Art, Spezialfall, Klasse). Die Funktionalität wird über die Schnittstelle geliefert. Das → Datenabstraktionsprinzip nutzt Information Hiding für einen bestimmten Zweck.

Klassen

Abstrakte Datentypmodule, die mittels der Vererbungsbeziehung (→ Vererbungsbeziehung) miteinander in Beziehung gebracht werden können. Sie sind somit i.a. in Vererbungshierarchien eingebettet. Die Schnittstelle einer Klasse in C++ wird in den Public-, Protected- und Private-Teil unterteilt. Der Public-Teil ist für alle importierenden Bausteine sichtbar, der Protected-Teil nur für Unterklassen einer Klasse. Der Private-Teil ist nur der Klassenrealisierung (dem Rumpf der Klasse) selbst zugänglich.

Kommentare

Spezielle lexikalische Einheiten von Programmiersprachen, die vom Compiler überlesen werden. In C++ beginnen Sie mit `//` und gehen bis zum Zeilenende. Alternativ wird ein (mehrzeiliger) Kommentar in `/* */` eingeschlossen (in Modula-2 sind Kommentare durch `(*` und `*)` eingerahmt). Kommentare sollten unbedingt in einem Programm verwendet werden. Sie dienen dem Festhalten von Lösungsentscheidungen und Lösungsideen. Kommentare können vor oder nach einem Programmteil stehen oder quellzeilenbegleitend sein.

Komplexität

Komplexitätsbetrachtungen untersuchen das Ergebnis der Programmentwicklung (Produktkomplexität). Die Komplexität des Entwicklungsprozesses wird i. d. R. nicht betrachtet. Meist beschränkt man sich auf die Betrachtung der Laufzeit– und der Speicherplatzkomplexität. Diese werden zur Programmlaufzeit gemessen (Monitoring) bzw. berechnet. Letzteres ist schwierig und Gegenstand der Komplexitätsuntersuchungen. Deshalb beschränkt man sich oft auf die Angabe von oberen Schranken ohne Berechnung ihrer inhärenten Konstanten (O–Notation in der Vorlesung).

Konstantendeklaration

Gibt es in C++ und Modula-2. Durch sie werden für einen compilezeitbestimmten Wert (Literal oder durch Literale gebildeter Ausdruck unter Nutzung weiterer Konstanten) ein Bezeichner eingeführt. Dies dient der Lesbarkeit und der Wartbarkeit von Programmen.

Konstrukte von Programmiersprachen

Konstrukte von Programmiersprachen sind die in einer jeweiligen Sprache eingebetteten Hilfsmittel zur Strukurierung von Programmen. Diese werden unterschieden in solche für die Ablaufkontrolle (Kontrollstrukturen), diejenigen zur Strukturierung von Daten (Datentypkonstruktoren) sowie solche zur Strukturierung der Gesamtstruktur (Modularisierungskonstrukte).

Kontrollierte Sprünge

Durch `break` können in C++ Schleifen (und nur diese) vorzeitig verlassen werden. Hierzu dient in Modula-2 die `Exit`-Anweisung, die sogar für das Verlassen beliebiger Kontrollstrukturen dient. In C++ gibt es ferner die Möglichkeit durch `continue` einen Schleifendurchlauf abzubrechen, um den nächsten Durchlauf zu beginnen.

Kontrollstrukturen

Kontrollstrukturen (→ Konstrukte von Programmiersprachen) dienen der Festlegung der Ablaufkontrolle. Hierzu zählen Anweisungssequenz, Fallunterscheidungen (bedingte Anweisung, Auswahlanweisung), Schleifen (Zählschleife, bedingte Schleifen) sowie Formen von Sprüngen. Die Kontrollstrukturen einer Programmiersprache ergeben sich aus den verschiedenen Formen für Anweisungen der Sprache.

Kurzform-Operatoren

Arithmetische Operatoren in Kurzform in C++ zur Abkürzung von binären Operationen und anschließender Zuweisung. (z.B. +=)

Kurzschluß-Auswertung

Für Boolesche Ausdrücke, die nicht vollständig ausgewertet werden, wenn der Wert des Gesamtausdrucks bereits feststeht. Oft stellt der erste Teilausdruck sicher, daß weitere überhaupt ausgewertet werden können.

Laufzeitkeller

Bei gegenseitigem Aufruf von Unterprogrammen (insbesondere bei rekursiven) ergibt sich zur Laufzeit ein kellerartig organisierter, dynamischer Datenspeicheranteil. Dieser enthält neben Formalparametern (call by value) und lokalen Variablen des Unterprogramms auch organisatorische Information für das Programmiersystem (sog. statischer und dynamischer Link). Dieser Teil des dynamischen Speicherbereichs ist deshalb nach dem Kellerprinzip organisiert, weil der Speicherbereich der zuletzt aufgerufenen Prozedur nach deren Ende als erster freigegeben werden kann.

Lexikalische Einheiten

Zusammenfassung der Syntax auf unterster Ebene von Zeichen zu sogenannten lexikalischen Einheiten (Tokens). Lexikalische Einheiten sind Bezeichner, Wortsymbole, Begrenzer, Literale und Kommentare.

Library-Files

Zusammenfassung von Header-Files vordefinierter Bausteine.

Listen

Standard-Datenstrukturen, die an vielen Stellen benötigt werden. Bei linearen Listen wird eine totale Ordnung der Listenelemente angenommen. Spezielle lineare Listen erlauben den Zugriff nur an den Enden der linearen Liste (Keller oder Stapel; Puffer oder Schlange). Lineare Listen können auf verschiedene Weise implementiert werden (sequentielle oder verkettete Realisierung). Die Verkettung kann über Zeiger oder Indizes (Cursor-Realisierung) erfolgen. Letzteres wird dann genutzt, wenn die Programmiersprache keine Zeiger besitzt. Man unterscheidet einfach verkettete Realisierung (Zeiger/

Indizes nur in eine Richtung) von mehrfach verketteter (in beide Richtungen). Bei sequentieller Realisierung wird die logische Reihenfolge (Hintereinanderreihenfolge) in einem sequentiellen Speichermedium (Feld, Datei) abgebildet. Bei der zirkulären Speicherung wird der sequentielle Speicher als geschlossen angenommen (vgl. zirkuläre Realisierung eines Puffers in der Vorlesung).

Literale

dienen dem direkten Hinschreiben eines konstanten Werts in einem Programm. Wir unterscheiden numerische Literale, Zeichen– und Zeichenkettenliterale. Numerische Literale werden in ganzzahlige Literale und reelle Literale unterschieden. Für erstere gibt es in C++ und Modula–2 oktale, dezimale und sedezimale ganzzahlige Literale.

Mengen

Für Teilmengen einer festen Trägermenge bietet Modula–2 einen Datentypkonstruktor `SET OF`. Kardinalitätseinschränkungen der Trägermenge sind überlicherweise 16, 32 o. ä. Intern wird charakteristische Speicherung benutzt, d.h. Ablage der Zugehörigkeitsinformation eines Elements zur Teilmenge. Dies führt intern zur Handhabung durch Bitvektoren. Die Mengenoperatoren werden so zu logischen Operatoren auf Bitvektoren. Für Mengen, die sich zur Laufzeit verändern und keine feste Trägermenge besitzen, sind obige Mengen nicht geeignet. Hier muß ein eigener Datentyp gebildet werden (→ abstrakter Datentyp), der über Listen, Bäume etc. realisiert wird.

Methode

Operation einer → Klasse

Modul

Ein Modul ist ein Baustein eines Softwaresystems. Er steht in Verbindung zu anderen Bausteinen. Die Exportschnittstelle gibt an, welche Dienste (Typen und Operationen) der Baustein offeriert. Die Importschnittstelle legt fest, welche Dienste anderer Bausteine zur Realisierung des Bausteins genutzt werden dürfen und müssen. Module dienen der getrennten Bearbeitung in einem Softwaresystem. Module müssen auf entsprechende Einheiten der Programmiersprache abgebildet werden (→ Modularisierungskontrukt). Module können zu Bibiliotheken zusammengefaßt werden. Sie stellen Einheiten der Wiederverwendung dar. Die Gesamtheit aller Bausteine (Module wie auch Teilsysteme) und ihrer Beziehungen ist in der Softwarearchitektur (Bauplan, Entwurfsspezifikation) festgehalten. Module sollten das Prinzip des Information Hiding beachten, d.h. daß Realisierungsdetails außerhalb des Moduls nicht sichtbar sind. Inwieweit die zugrundeliegende Programmiersprache dieses unterstützt, hängt von deren Modularisierungskonstrukten ab. Modula-2 kennt Module, die dort in Form zweier Einheiten, nämlich Schnittstelle (definition module) und Rumpf (implementation module) festgehalten werden. C++ kennt keine expliziten Modularisierungskonstrukte bis auf Klassen. Andere Bausteine müssen auf Dateiverwaltungsebene (→ Header- und → Implementation Files) gehandhabt werden.

Modula−2

Programmiersprache klassischer Bauart (imperativ, prozedural) mit Softwaretechnikeinfluß (Modularisierungskonstrukte). Modula–2 ist gut für die Ausbildung geeignet, in der Industrie und Programmentwicklung weniger verbreitet. Modula–2 wurde von N.Wirth in den 80er Jahren definiert. Auf PCs stehen Public Domain Compiler zur Verfügung. Modula–3, der Nachfolger von Modula–2, besitzt auch Konstrukte für die Objektorientierung.

Modularisierungskonstrukte

Konstrukte (→ Konstrukte von Programmiersprachen) zur Festlegung der Grobstruktur eines Programms. Hierzu zählen Prozeduren, Module, Klassen sowie die Möglichkeit der Bildung größerer Einheiten mit ihnen (Teilsysteme).

offene Felder

für Formalparameter von Unterprogrammen in Modula–2. Diese Felder müssen eindimensional sein. Innerhalb der Prozedur beginnt der Index des Aktualparameters bei 0. Die obere Grenze ist nicht festgelegt. Offene Felder dienen unter obigen Einschränkungen zur Formulierung von Unterprogrammen, die unabhängig von den Indexgrenzen und damit der Komponentenzahl des Feldes sind.

Parameterzuordnung (s−Mechanismus)

Mechanismus zur Zuordnung (Übergabe) von Aktualparametern zu Formalparametern bei Unterprogrammen. Die Üblichen sind call by value (Aufruf über den Wert) und call by reference (Aufruf über die Adresse). Im ersten Fall fungiert der Formalparameter als lokale Variable. Der Wert des Aktualparameters wird kopiert. Solche Parameter dienen zur Eingabe von Werten in das Unterprogramm (Eingangsparameter). Im zweiten Fall wird der Aktualparameter (seine

Adresse) zur Bindung herangezogen (direkte Veränderung des Aktualparameters). Diese zweite Art dient der Handhabung von Ein–/Ausgabeparametern (Transienten) und Ausgabeparametern.

Pascal

Ausbildungssprache die etwa 1970 von N. Wirth definiert wurde. Pascal besitzt keine Hilfsmittel für das Programmieren im Großen. Die Kontrollstrukturen und Datentypkonstruktoren von Pascal sind mittlerweile Standard in klassischen Programmiersprachen. Pascaldialekte besitzen zahlreiche Erweiterungen in allerdings nicht standardisierter Form.

Portabilität

nennt man die Eigenschaft eines Programmsystems, unabhänig von der konkreten Plattform (Hardware, Programmiersystem, Dateiverwaltungssystem, Ein–/Ausgabesystem etc. und deren Versionen) zu sein.

Prinzipien → Strategie

Programmiersprachenkonstrukte

→Konstrukte von Programmiersprachen

Programmieren im Großen

Systematische Vorgehensweise bei der Gestaltung großer Programmsysteme. Hierzu ist ein Bauplan (eine Architektur) zu erstellen, die Export– und Importschnittstellen der Bausteine sind festzulegen. Die Realisierung der Bausteine ist Gegenstand des Programmierens im Kleinen. In der Vorlesung taucht Programmieren im Großen nur andeutungsweise auf. Die Schnittstelle festgelegter Bausteine erfolgt normalerweise allein auf syntaktischer Ebene, d.h. der Festlegung des Typs und der Reihenfolge der Formalparameter von Operationen der Bausteine.

Programmieren im Kleinen

Systematische Vorgehensweise bei der Ausgestaltung kleiner Programme oder der Ausgestaltung von Programmbausteinen bzw. die Ausgestaltung selbst. Somit faßt der Begriff die methodische Vorgehensweise als auch die Detailrealisierung solcher Bausteine zusammen.

Prototyp

Spezielle C++-Terminologie für Funktionskopf.

Quicksort

Verbessertes Sortierverfahren nach der Strategie des "Sortierens durch Vertauschen". Es wird häufig verwendet. Allerdings ist Quicksort im schlechtesten Fall von quadratischem Aufwand.

Records → Verbunde

reelle Datentypen

(`float`, `double` in C, `FLOAT` in Modula–2,) für numerisch reelle Datentypen (Zahlenwerte) mit den üblichen arithmetischen Operationen.

Referenzen

Eine spezielle Form von Zeigern in C++. Solche Zeiger müssen bei der Deklaration gesetzt werden und werden nicht verändert. Bei Zugriffen muß nicht dereferenziert werden. Auf diese Weise können Zugriffswege auf Datenobjekte über Adressen eingerichtet werden.

Reihungen → Felder

Robustheit

nennt man die Eigenschaft eines Programmsystems, wenn falsche Bedienereingaben vollständig abgeprüft werden und somit nicht zur Programmbeendigung oder sogar zum Programmabsturz führen.

Rumpf eines Moduls

Der Rumpf eines Moduls (→ Modul) realisiert die Exportschnittstelle unter Nutzung der Dienste (Ressourcen) der Importschnittstelle. Hierzu werden geeignete Datenstrukturen (Deklarationsteil) und Ablaufstrukturen (Anweisungsteil) festgelegt. Ggfl. werden lokale Prozeduren im Deklarationsteil definiert, wenn diese nicht groß sind oder nicht anderweitig verwendbar sein sollen. In letzterem Falle werden sie selbst zu Modulen.

Schleifen

dienen zur wiederholten Ausführung einer Anweisungsfolge (des Schleifenrumpfes). Wir unterscheiden Zählschleifen und bedingte Schleifen. Letztere werden wiederum unterschieden in while–Schleifen (Fortsetzungsbedingung, die vor dem Schleifenrumpf ausgewertet wird) und until–Schleifen (Abbruchbedingung, die nach dem Schleifenrumpf ausgewertet wird). Bei bedingten Schleifen ist unbedingt auf Termination zu achten. Hierzu wird in den Terminationsüberlegungen eine Monotonieüberlegung angestellt und eine Schranke gefunden, die erreicht werden muß.

Schnittstelle eines Moduls

Die Schnittstelle eines Moduls (→ Modul) besteht aus der Export- und der Importschnittstelle. Manchmal wird auch nur die erstere als Schnittstelle bezeichnet.

schrittweise Verfeinerung

nennt man die Programmentwicklungs–Methode, über Pseudoanweisungen (abstrakte Anweisungen) einen Anweisungsteil schrittweise zu verfeinern, bis man auf der Ebene der Programmiersprache angelangt ist. Die Pseudoanweisungen bleiben gegebenenfalls als Kommentare zurück.

Seiteneffekt(freiheit)

Ausdrücke sollten einen Wert zurückliefern und bei ihrer Auswertung keine Seiteneffekte erzeugen. Dies bedeutet z.B., daß in C++ keine ++–Operatoren o. ä. in Ausdrücken verwendet werden sollten. Ebenso sollten Funktionen keine Seiteneffekte erzeugen (z.B. Verändern globaler Variablen).

selbstdefinierte Datentypen → Typ

Sichtbarkeitsbereich

ist ein Teilbereich des Gültigkeitsbereichs, in dem eine Deklaration angesprochen werden kann. Der Sichtbarkeitsbereich ist ein Teilbereich, da Verdeckung von Variablen, Unterprogrammen (Überladung in C++) auftreten kann. In C++ kann eine verdeckte Deklaration durch den Scope-Operator `(::)` angesprochen werden, bei überladenen Unterprogrammen sucht der Compiler das richtige heraus.

Simulation von Modulen in C++

Die Schnittstelle eines Moduls wird in C++ in einem Header-File zusammengefaßt, die Realisierung des Modulrumpfes in einem Implementation-File. Dies ist zunächst ähnlich zu Modula-2 (Definition und Implementation Module). Zwischen beiden Dateien erfolgt in C++ aber keine Überprüfung auf Konsistenz. Bei Importen (Einbeziehen anderer Header-Files über `#ifndef` und `#define` erfolgt in C++ kein Import auf der Ebene einzelner Ressourcen eines Moduls. Bei sehr disziplinierter Verwendung von Header-Files, Kommentierung und disziplinierter Programmierung von Modulen unter Bezugnahme auf die Header-Files anderer Module können in C++ nicht vorhandene Modularisierungskonstrukte simuliert werden. Dies wird in der Vorlesung gezeigt (Funktionsmodule, Datenobjekt- und Datentypmodule sowie ihr Import für weitere Module).

skalare Datentypen → Typ

Sortieren

nennt man das Anordnen von Datenelementen nach einem Teil ihrer Information (Schlüssel oder Primärschlüssel). Die Werte des Schlüssels müssen total geordnet sein. Dies ist für ganzzahlige Werte der Fall bzw. für Zeichenwerte. Bei zusammengesetzten Schlüsseln wird lexikographische Ordnung herangezogen. Sortieren wird unterschieden in batchartiges oder inkrementelles, internes oder externes. In der Vorlesung haben wir direkte Sortierverfahren (einfache Verfahren) kennengelernt, z.B. Bubblesort. Quicksort wurde als Beispiel eines verbesserten Verfahrens behandelt (im Mittel O(n logn).

stepwise refinement

→ schrittweise Verfeinerung

Strategien (Prinzipien für Algorithmen)

Strategien fassen bestimmte Algorithmen zu einer Klasse zusammen. Alle Repräsentanten der Klasse folgen der gleichen Strategie. Beispielsweise gibt es für ein Sortierverfahen 4 Strategien (Einfügen, Austauschen, Auswählen, Mischen). Analoges gilt für Suchverfahren. Durch Festlegung von Details wird aus einer Strategie ein konkretes Verfahren. In der Vorlesung haben wir z.B. zwei Verfahren (Bubblesort, Quicksort) kennengelernt, die der Strategie "Sortieren durch Vertauschen" folgen. Andere Klasseneinteilungen für Sortierverfahren sind intern oder extern, direkte oder verbesserte Verfahren, stabile oder instabile Verfahren, batchartige oder inkrementelle Verfahren.

Struktogramm

Graphische Darstellung für Programme. Dabei wird nur die Ablaufkontrolle dargestellt. Für die üblichen Kontrollstrukturen gibt es entsprechende graphische Elemente (Sequenz, bedingte Anweisung, Auswahlanweisung, Schleifen, Prozeduren).
→ AnhangII

Strukturen → Verbunde

Suchen

nennt man ein Programm zum Auffinden von Datenelementen, wenn deren Schlüssel bekannt ist. Sortieren ist eine Voraussetzung für effizientes Suchen, um erschöpfende Suche zu vermeiden. Bei binärer Suche wird fortgesetzte Intervallschachtelung angewendet. Binäre (ausgeglichene) Suchbäume sind eine Technik zur einfachen Realisierung binärer Suche auf i. d. R. veränderlichem Datenbestand.

Syntaxnotation

Formale Festlegung der Syntax einer Programmiersprache. Hierzu bedarf es einer formalen Sprache zur Festlegung dieser Syntax. In der Vorlesung wird lediglich die kontextfreie Syntax formal durch EBNF–Regeln oder durch Syntaxdiagramme definiert. Kontextsensitive Syntax wird bei Programmiersprachendefinitionen in der Regel durch Prosa erläutert.

Test

Experimentelle Überprüfung eines Programms. Test wird gegliedert in Testdatengewinnung, Testvorbereitung, Testdurchführung und Testauswertung. Der Modultest dient der Überprüfung einzelner Bausteine, der Integrationstest der Überprüfung von Gruppen von Bausteinen, der Systemtest der des gesamten Programms. In letzterem Fall werden auch Funktions- und Leistungsüberprüfung betrachtet. Gewisse Schritte des Tests sind automatisierbar oder teilweise automatisierbar (Testdatengewinnung, Erzeugen Teststummel und Testtreiber, Durchführung von Regressionstest). Durch Testen kann nur die Anwesenheit, aber nicht die Abwesenheit von Fehlern festgestellt werden (Dijkstra)

Trace

→ s. Ablaufverfolgung

Typ

Datentypen unterscheidet man in vordefinierte oder selbstdefinierte. Eine andere Einteilung unterscheidet einfache und zusammengesetzte. Programmiersprachen besitzen in der Regel sowohl einfache als auch zusammengesetzte vordefinierte Datentypen, die meisten der vordefinierten Datentypen sind einfache Datentypen (skalare Datentypen). Ein Typ faßt die Struktur einer Klasse von Objekten zusammen (Klasse hier nicht im objektorientierten Sinne): Struktur, Wertebereich, sowie deren Operationen. Operationen sind oft implizit, z.B. `+` für `int`.

Typgleichheit/Typäquivalenz

Hierunter versteht man die Regeln einer Programmiersprache, die festlegen, wann zwei Objekte vom Typ her als gleich zu betrachten sind. Bei Programmiersprachen mit Strukturäquivalenz wird die Struktur von Typdefinitionen untersucht. Modula–2 führt keine Strukturäquivalenz ein. Jede Typdefinition (Strukturfestlegung eines Typs) führt einen neuen Typ ein! Mit einer Synonymdeklaration können verschiedene Datentypen eingeführt werden, die als äquivalent betrachtet werden.

Überladung von Prozeduren

Gibt es nur in C++. Damit sind verschiedene Unterprogramme mit gleichem Namen definierbar, die sich anhand des Parametertyp-Profils (Anzahl, Reihenfolge, Typ der Formalparameter) unterscheiden müssen. In der kontextsensitiven Syntaxanalyse wird das passende Unterprogramm herausgesucht, das dabei eindeutig definierbar sein muß.

Unterbereichstypen

In Modula–2 für skalare Datentypen möglich zur Festlegung eines laufzeitabhängigen Bereichs. Unterbereichstypen passen nicht in das sonstige Typkonzept, da sie keine strukturellen Festlegungen treffen, sondern nur Laufzeiteinschränkungen wiedergeben.

Unterprogramme

Hilfsmittel zur Grobstrukturierung von Programmen (→ Modularisierungskonstrukte). Unterprogramme bestehen aus einem Unterprogrammkopf (Unterprogrammschnittstelle mit Formalparameterliste). Unterprogramme liefern einen Wert (Funktion) oder haben den Charakter einer zusammengesetzten Anweisung (Prozeduren, Verfahrensprozeduren). Im Unterprogramm–Rumpf wird die Berechnung ausformuliert. Hierzu können Deklarationen eingeführt werden und der Anweisungsteil kann beliebig strukturiert sein. Funktionen bzw. Prozeduren sind in C++ als auch in Modula-2 durch die Form des Unterprogrammkopfes unterscheidbar.

variante Verbunde

Variante Verbunde sind Verbunde (→ Verbunde) mit einem gemeinsamen Teil und unterschiedlichen Varianten. Variantenauswahl wird durch die Diskriminante, meist eine Variable eines Aufzählungstyps, gesteuert. Die verschiedenen Varianten können unterschiedlich strukturiert sein und somit auch unterschiedlich lang sein. Die Diskriminante ist Teil des Verbunds. Der Compiler legt soviel Platz an, wie der längste Variantenteil erfordert.

Verbunde (Strukturen, Records)

Zusammenfassung verschiedenartiger Datenobjekte zu einem Ganzen. Zugriff über Selektornamen. Der Selektorpfad steht zur Programmerstellungszeit fest. Man unterscheidet einfache Verbunde von varianten Verbunden. Bei varianten Verbunden folgen einem gemeinsamen Teil unterschiedliche Varianten. Ein konkretes Datenobjekt kann zu einem bestimmten Zeitpunkt der Programmausführung nur einer Variante angehören.

Vererbungsbeziehung

Aus Klassen können Spezialisierungen gewonnen werden. Diese Beziehung heißt Spezialisierung oder Vererbung, da die Unterklasse von der Oberklasse deren Eigen-

schaften erbt. Die umgekehrte Beziehung heißt Generalisierung. Klassen werden in Vererbungshierarchien angeordnet, um so die Ähnlichkeit zwischen verschiedenen Klassen ausdrücken zu können. Es ist i.a. schwierig, saubere Vererbungshierarchien aufzubauen. Falls eine Klasse nur eine Oberklasse haben darf, spricht man von Einfachvererbung, ansonsten von Mehrfachvererbung. Objektorientierte Softwarekonstruktion heißt, daß ein Softwaresystem oder ein Teil hiervon nur aus Vererbungshierarchien aufgebaut wird.

vordefinierte Datentypen → Typ

White-Box-Test

Modultest unter Betrachtung der internen Realisierung eines Bausteins. Üblich sind sogen. Überdeckungsverfahren. In der Vorlesung haben wir die Flußgraphenmethode kennengelernt, welche die sämtlich möglichen Pfade der Ablaufkontrolle betrachtet. Zur Reduktion der Testfälle werden dabei mehrmalige Schleifendurchläufe zu einem Muster zusammengefaßt. Die entsprechenden möglichen Ablaufmuster werden einzeln durch Testfälle betrachtet.

wohlstrukturierte Programme

wohlstrukturierte Programme besitzen keine explizite Sprunganweisung. Sie sind ausschließlich mittels Sequenz, Fallunterscheidung (`if, switch`), Iteration (`for, while`) gebildet. In der sogenannten goto–Kontroverse wurde das Für und das Wider von Sprunganweisungen ausführlich und zum Teil ideologisch diskutiert. Kontrollierte Sprünge sind Sprunganweisungen, die keine weiteren, als die gegebenen Sprungmarken der Kontrollstrukturen einführen. Verwendet man diese, so spricht man auch von `EXIT`–wohlstrukturierten Programmen.

Wortsymbole

Bezeichner mit vordefinierter Semantik in einer Programmiersprache. Meist dürfen sie, in jedem Falle sollten sie nicht für selbstdefinierte Bezeichner verwendet werden (Beispiele `if, do, const` in C++; `IF, TYPE` in Modula-2).

Zählschleifen

Eine Form von Schleifen (→ Kontrollstrukturen, → Anweisungen) zur wiederholten Ausführung eines Programmteils, des Rumpfs der Schleife, der eine Anweisungsfolge darstellt. Zählschleifen haben allgemein kein Terminationsproblem. Im Schleifenkopf sind Anfangswerte, Endwerte und ggfl. Schrittweite angegeben. Alle Werte können laufzeitabhängig sein.

Zeichenketten

werden als Felder mit Komponenten des Typs `char` in C++ bzw. `Character` in Modula–2 betrachtet. Zeichenketten können durch Zeichenkettenliterale dargestellt sein bzw. durch Zeichenkettenausdrücke mithilfe des Konkatenationsoperators gebildet werden.

Zeichentypen

`char` in C, `CHAR` in Modula–2 mit den Zeichen des jeweiligen, zugrundeliegenden Alphabets.

Zeiger

Mithilfe von Zeigertypen (→ Datentypkonstruktoren) können beliebige vernetzte Datentypen deklariert werden. Diese dienen einerseits dazu, beliebige Relationen zwischen Datenobjekten festlegen zu können. Zum anderen können rekursive Datentypen damit definiert werden (z.B. Bäume). Zeiger sind in Modula–2 typisiert, d.h. sie können nur auf Objekte eines bestimmten Typs verweisen. Zeiger werden zur Laufzeit neu eingerichtet bei der Erzeugung von Haldenobjekten (Allokation, `new` in Modula-2) oder durch Setzen in einer Wertzuweisung. Durch Löschen von Haldenobjekten können hängende Zeiger entstehen (dangling references). Mit Haldenobjekten, Zeigern auf Haldenobjekten und Verweisen zwischen Haldenobjekten über Zeiger können beliebig komplexe Datenstrukturen gehandhabt werden. Solche Strukturen sollten möglichst modullokal gehalten werden. Neben hängenden Zeigern und unübersichtlichen Datenstrukturen ergeben sich Probleme mit verschiedenen Zugriffspfaden (Aliasing) sowie nicht mehr zugreifbaren Haldenobjekten.

zusammengesetzte Datentypen → Typ