Informatik I

Programmiersprache C++
(Zusammenfassung)

erstellt durch:

Simon Eugster D-MAVT

SS 2005

1. Elementare Konstrukte

1.1 Header eines Programms

```
#include<iostream>      //Zugang zu (vordefinierten) Hilfsprogrammen (Hier: Ein- und
                        //Ausgaberoutinen (cin, cout).
using namespace std;    //anstelle von „std::cout <<“ kann jetzt nur noch „cout <<“
                        //geschrieben werden.
```

```
#include<iostream>
using namespace std;

int main(){

cout << „Hello World“ << endl;

}
```

1.2 Variablendeklaration

Typname	Variablenname-Liste

Typname: int, double, float, bool…

```
int x;
int a, b = 1, c, d = 2;
bool z = false;
```

1.3 Wertzuweisung

Variablenname	=	Ausdruck

```
x = 1;
x = x + 1;
x = y * z;
```

1.4 Datentypen

1.4.1 (signed) int:

- ganze Zahlen
- Bereich: -2147483648 bis 2147483648 (2^{32} Werte)
- Schreibweise 0, -10, 2341
- Operatoren
  - + Addition
  - – Subtraktion
  - * Multiplikation
  - / ganzzahligie Division
  - % Divisionsrest
- Ganzzahlige Division 15 / 6 = 2 und 15 % 6 = 3
- Modulares Rechnen: 2147483648+1 = -2147483648
- unsigned int (Bereich 0 bis 4294967295), short int 2^{16}, long int 2^{32}.

1.4.2 float (4 Byte), double (8 Byte):

- modelliert reelle Zahlen
- Schreibweise: 3.14, -1.5, .3, -1.1E-5, 1.1e6
- Ohne Dezimalpunkt oder E (z.B. 4) : Typ int ➔ 3.0: Typ double
- 3.0f, 3.0F: Typ float
- Operationen: +, -, *, / weitere Operationen per #include<math>

1.4.3 char

- Buchstaben, Ziffern, Zeichen, etc.
- Im Programm char c; c = ´A´; oder char c; c = 65;
- char werden als ganze Zahlen behandelt zwischen 0 und 255
- Arithmetik möglich z.B.: ´A´ + 20 (gleich ´U´)

Zeilenende	\n
horizontaler Tabulator	\t
vertikaler Tabulator	\v
Wagenrücklauf	\r
Alarm (Piepston)	\a
\ Backslash	\\
` Häkchen	\`
Anführungszeichen	\¨

ASCII-Tabelle
→ A, B, ..., Z : 65, 66, ..., 90.
→ a, b, ..., z : 97, 98, ..., 122.
→ 0, 1, ..., 9 : 48, 49, ..., 57.

! Achtung Häkchen nicht vergessen bei cout << (cout << ´A´) !

1.4.4 bool

- Wahrheitswert true (≠ 0), false (0)
- Verknüpfung: logische Und (&&), Oder (| |), Negation (!).

x	y	x&&y
true	true	true
false	true	false
true	false	false
false	false	false

x	y	x || y
true	true	true
false	true	true
true	false	true
false	false	false

x	!x
true	false
false	true

- Vergleichsoperatoren
  - < kleiner Relation
  - > grösser Relation
  - <= kleiner-gleich Relation
  - >= grösser-gleich Relation
  - = = gleich Relation
  - != ungleich Relation

```
x = 5, y = 7. z = true
(x<y) | | (y == 0)        ➔ ergibt true || false = true
```

1.4.5 Typumwandlung

- in C++: d = i; möglich wobei d von Typ double, i von Typ int. ➔ i wird in double umgewandelt.
- i = d; d wird gerundet, Compiler liefert Warnung
- explizite Typumwandlung (cast):

  Typname (Ausdruck) `double (3)`

  (Typname) Ausdruck `(double) 3`

1.5 Binärrechnen (Anhand von Beispielen)

Umrechnen vom Binär- ins Dezimalsystem:
1011 binär: $1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 8 + 0 + 2 + 1 = 11$ dezimal

Umrechnen vom Dezimal- ins Binärsystem:

Zahl : 2	Rest
14	
7	0
3	1
1	1
0	1

➔ 1110 binär

2 Kontrollstrukturen

2.1 if-Anweisung

- if (*Bedingung*) *Block*
- *Block* wird dann ausgeführt, wenn *Bedingung* zu true auswertet
- Allgemein: if (*Bedingung*) *B1* else *B2*
- *Block* besteht aus: *Anweisung*; oder {*Anweisungsfolge*;}
- else bezieht sich immer auf des letztgenannte if, zu dem noch kein else gehört.
- `x = Bedingung ? Ausdruck#1 : Ausdruck#2;`
  ➔ Kurzform für:

```
if (Bedingung)
     x = Ausdruck#1;
else x = Ausdruck#2;
```

2.2 switch-Anweisung

```
switch (Ausdruck) {
    case Konstante #1: Anweisungsfolge #1; break;
    .
    .
   case Konstante #n: Anweisungsfolge #n; break;
   default: Anweisungsfolge #n+1;
}
```

- Ist *Ausdruck* gleich *Konstante #i*, so wird *Anweisungsfolge #i* ausgeführt.
- Ist *Ausdruck* ungleich *Konstante #*i für alle i, so wird *Anweisungsfolge #n+1* ausgeführt.
- *Ausdruck* muss eine ganze Zahl ergeben.
- default kann fehlen
- fehlt break, kann auch die nächste *Anweisungsfolge* ausgeführt werden.

2.3 for-Schleife

- for (*Initialisierung* ; *Bedingung* ; *Anhang*) *Block*
- *Initialisierung* wird genau einmal vor dem erstem Durchlauf der Schleife ausgeführt.
- *Bedingung* wird vor jedem Durchlauf ausgewertet. Wert false ➔ Verlassen der Schleife.
- Wurde Schleife nicht verlassen, so wird *Block* ausgeführt.
- Am Ende jedes Durchlaufs wir *Anhang* ausgeführt.
- Ist *Bedingung* leer ➔ Endlosschleife

```
for (i = 1; 1 <= n; i = i +1){
      summe = summe + i;
}
```

Abkürzungen:

`summe += i;` entspricht: summe = summe + i; (Ebenso: -=, *=,…)

`i++;` entspricht: i = i +1; (Ebenso: i--)

wobei zu beachten ist, dass dies erst nach durchgeführter Aktion gemacht wird.

```
i=2;
d = 4*i++;
cout << d;  // d = 8
cout << i;  // i = 3
```

++i; erhöht i um 1 bevor der Ausdruck ausgeführt wird

```
i=2;
d = 4*++i;
cout << d;  // d = 12
cout << i;  // i = 3
```

Kleiner-Gleich-Bedingungen in for-Schleifen: a < b ≤ n

```
for(a=0;a<n;a++){
      for(b=a+1;b<=n;b++){}
}
```

2.4 while-Schleife

- while (*Bedingung*) *Block*;
- Durchlauf des *Blocks* wenn *Bedingung* zu true auswertet.

2.5 do-Schleife

- do
  Block
  while (*Bedingung*);
- *Block* wird so oft wiederholt, bis *Bedingung* zu false auswertet. (min. 1 Durchlauf)

→ break verlässt eine Schleife unmittelbar (Geschachtelte Schleifen: aktuelle Schleife).
→ continue überspringt den Rest des Rumpfs der Schleife und geht zur nächsten Auswertung der Bedingung.

3 Abgeleitete Typen

3.1 Arrays

- *Typname Variablenname* [*Feldgrösse*]
- *Feldgrösse* muss Konstante w sein
- Zugriff: *Variablenname*[0], ..., *Variablenname*[w-1]
- ```
  int i = 6;
  double a[10];
  a[5]= 7.5;
  a[i] = 2 * a[i-1];
  ```
  → a[5] enthält Wert 7.5
  → a[6] enthält Wert 15
- Konstanten
  ```
  const int max_groesse = 10;
  int a[max_groesse];
  ```
- Dynamische Deklaration (➔ Siehe Pointer)
  ```
  int a[n];
  a = new int[n];
  ```
- Mehrdimensionale Arrays möglich: double a[10][10] oder double a[10] [10] [10]
  ```

- Initialisierung möglich:
  ```
  int a[5]= {1,2,3,4,5};
  int a[5]= {1,2,3};   ➔ a[3] und a[4] werden mit 0 initialisiert
  int a[] = {1,2,3,4,5};  ➔ Grösse wird auf 5 festgelegt
  int a[3][2] = {{1,2},{3,4},{5,6}};
  ```
- Arrays an Funktionen übergeben:
  `int f(int a[]){...}` ➔ Aufruf: `f(a);`

Structs

- Daten mit heterogenem Charakter die aber zusammengehören ➔ struct
- ein struct ist eine Art Datentyp für ein bestimmtes Problem
- struct *Name* { *Komponentenliste* };
- ```
  struct point {
      int x, y;
      double intensity ;
  } p ;
  ```
  ➔ p ist vom Typ struct point und man kann mit p.x, p.y und p.intensity auf die einzelnen Komponenten Zugreifen.
- Structs lassen sich auch Schachteln.
- Initialisierung: `struct point p = {12, 27, 0.75};`
- `p = q` kopiert vollständig
- Kopieren von Arrays mit Hilfe von structs
  ```
  struct array{
  int inhalt[10];
  } a,b;
  ```
- `a = b` kopiert jedes `a.inhalt[i]` in `b.inhalt[i]`

3.2 Unions

- ähnlich wie structs. Es wird jedoch nur ein Datum gespeichert.
- ```
  union Zahl{
      int ganz;
      double reell;
  } ;
  union Zahl z;
  z.ganz = 3;    //speichert 3 als int ab.
  z.reell = 3.14 //speichert z als double und überschreibt 3
  ```

3.3 Enums

- enum *Name* { *Identiferliste* };
- ```
  enum Farbe {rot, gruen, blau};  //einführen von Konstanten (rot = 0,
  gruen = 1, blau =2)
  enum Farbe f;
  f = rot;
  if (f == gruen) {...};
  ```

4 Datenstrukturen

4.1 Sichtbarkeit von Variablen

- Block wird durch geschweifte Klammern `{...}` begrenzt.
- Geltungsbereich eines Namens beginnt am Deklarationspunkt und endet am Blockende.
- Namen ausserhalb irgendeines Blocks sind globale Namen.
- Lokale (innere) Namen haben im Block Vorrang (Verdeckung: in einem inneren Block wird der gleiche Name nochmals deklariert).
  ```

- Bei Blockende gehen die lokalen Werte verloren.
- Zugriff auf globale Variable mit „ :: “

```
int x=7;
int main(){
int x=3;
cout << x; //x = 3
cout << ::x;     //x = 7
```

4.2 Funktionen

- *Typ Name* (*Parameterliste*) *Block*; ➔ Funktionskopf
  double sqrt (double x, int y) {...}
- Bei Aufruf werden die formalen Parameter durch aktuelle Parameter ersetzt.
- Formale Parameter
  - Werteparameter: call by value (Wert wird kopiert)
  - Variablenparamter: call by reference (Adresse wird kopiert)
- *Typ* legt Typ des Funktionswerts fest.
- Funktionswert wird mittels return zurückgegeben.
- return kann mehrmals vorkommen.
- Nach erstem ausgeführten return wird die Funktion verlassen.
- Funktion kann auch keinen Wert zurückliefern: Typ void.
- Signatur: weglassen der Parameternamen und des Blocks um anzukündigen, dass eine solche Funktion vorhanden ist; zu Beginn des Programms. Beispiel: `int f(int,double);`

4.3 Referenzen

- Referenzen ermöglichen es, für eine Variable mehrere Namen zu vergeben. (Adresse wird kopiert)
- Initialisierung:

```
int n;
int& nr = n;
```

  n und nr können nun als Synonyme verwendet werden.
- Call by Reference:
  `void f(int& y) {...y++...}` mit Funktionsaufruf `f(x)`
  Durch den Funktionsaufruf f(x) kann die Variable x verändert werden.
- Damit es zu keiner unbeabsichtigten Veränderung des Parameters durch die Funktion kommt, am besten Deklaration int f(const int& y) ➔ bei veränderung Compiler-Error.

4.4 Pointer

- Pointer (Zeiger) ist ein Datentyp, der eine Adresse im Hauptspeicher speichern kann
- Deklaration: `int* p;`
- Zugriff:
  `pointer` entspricht der Adresse des Pointers pointer
  `*pointer` entspricht dem Wert des Pointers pointer
  `&normal` entspricht der Adresse von int normal

```
int* pointer;
int normal;
normal = *pointer;   // t entspricht Wert des Pointers p
&normal = pointer;   // t erhält die Adresse von Pointer p
```

- Call by Reference:
  `void f(int* y) {...(*y)++...}` mit Funktionsaufruf `f(&x)`
- Pointer-Arithmetik:
  `p = p +2;`
  Der Pointer wird 2 mal die Anzahl vom Speicherzellen weitergeschoben, die zum Speichern des Basistyps von p benötigt werden.

- Nützlich, wenn man weiss, dass viele Werte hintereinander im Speicher stehen, wie bei Arrays.
- Arrays werden durch Pointer realisiert. int a[10]; deklariert a als Pointer auf int und reserviert dann 10 Speicherplätze. a zeigt auf den ersten Platz, also a[0].
- `a`: Pointer auf nullte Element des Arrays
  `a+i`: Pointer auf i-te Element
  `*(a+i)`: Wert des i-ten Element
- Dynamische Variablen:
  new *Typ* erzeugt neue Variable vom Typ *Typ* und liefert Adresse zurück, welche einem Pointer zugewiesen werden kann
  ```
  int* p = new int;
  int* p = new int[10];        //für pointer p werden 10 Speicherzellen reserviert.
  ```

4.5 Char

- `char* s = „string“;`
  reserviert 7 Speicherplätze und initialisiert s mit „string\0“
- Ausgabe: `while(*s!=0){cout<<*c<<;c++}`

4.6 Arithmetik

- Lineare Suche: N Schritte
- Vorsortierte Suche mit binärer Suche: $[\log_2 N]+1$ Durchläufe ➔Asymptotischer Verlauf

Definition

$f, g : \mathbb{N}_{>0} \to \mathbb{N}_{>0}$

- f wächst asymptotisch mindestens so schnell wie g, falls es eine Konstante $c > 0$ gibt, so daß $g(n) \leq cf(n)$ für alle n gilt. (Schreibweise: $g \leq O(f)$)
- f wächst asymptotisch echt schneller als g, falls g nicht asymptotisch mindestens so schnell wie f wächst.

4.7 Rekursion

- Funktionen dürfen andere Funktionen aufrufen ... auch sich selbst!
- wichtig: explizite Abbruchbedingung
- Aufbau: f(...) {*Rekursionsbasis* (Abbruchbedingung)
  return *Rekursionsschritt*}
- Fakultätsfunktion n!:
  ```
  int fak_rek(int n) {
      if (n == 0) return 1;
      return n * fak_rek(n-1);
  }
  ```
- Jede Rekursion kann durch Schleifen realisiert werden
- Iterativer Ansatz oftmals schneller
- Rekursiver Ansatz oftmals klarer (kürzerer Code)

5 Listen

5.1 Listen allgemein

- ```
  struct element {
      int key;
  };
  struct element* list = new element[N];
  ```
- Suchen, Einfügen und Löschen braucht jeweils O(N) Schritte
  ```

5.2 Verkettete Listen

- ```
  struct element {
      struct element* next;
      int key;
  };
  struct element* list;
  ```
- Komponente next verweist auf nächtes Element in der Liste
  Zugriff per `(*list).next` (Klammern sind notwendig)
  Alternativ: `list -> next`
- Listenende durch `next == 0` oder `NULL` angezeigt (Nullpointer)

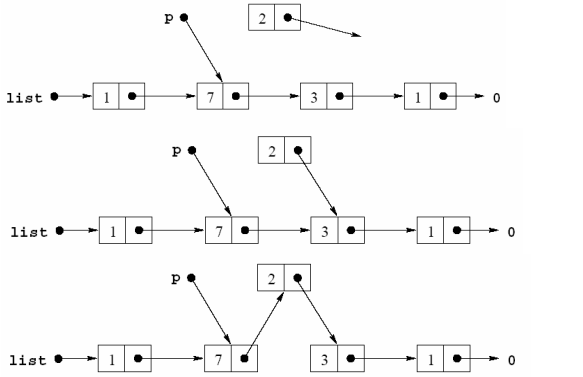
5.3 Listen Suche

```
p= list
while (p!=0){
  if (p->key == k) return p;
  p = p -> next;        //durchläuft Liste
}
return 0;
```

Benötigt O(N) Schritte

5.4 Listen Einfügen (Einfügen von Element struct element e an der Stelle p)

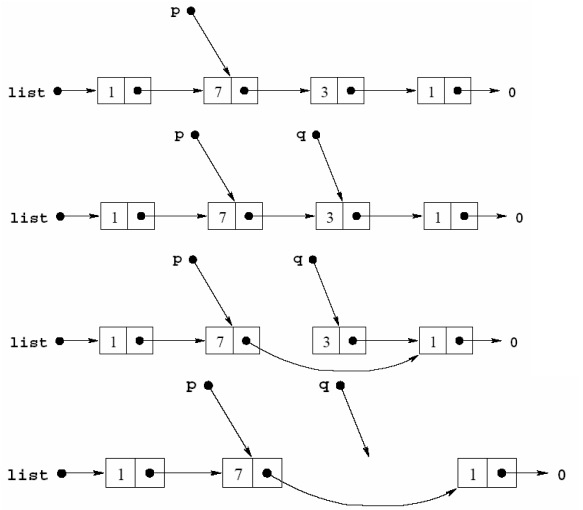
```
e.next = p-> next;
p->next = &e;
```



Benötigt 2 Schritte

5.5 Listen Löschen (Element löschen wo p->next hinzeigt)

```
q = p->next;
p->next = p->next->next;
delete q;
```



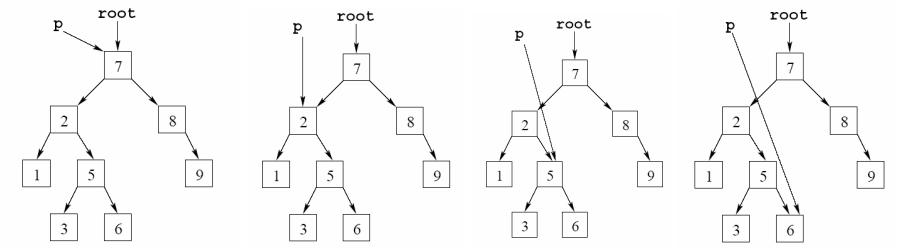
Benötigt 3 Schritte
```

6 Bäume

6.1 Bäume allgemein

- Jedes Element hat nun 2 Nachfolger
- ```
  struct element {
      struct element* left, right;
      int key;
  };
  ```
- Elemente heissen Knoten
- Sind die Komponenten `left` und `right` der Nullpointer so heisst der Knoten Blatt
- Startknoten heisst Wurzel (root)
- Höhe eines Baumes max. Abstand eines Blattes (= Anzahl Knoten) zur Wurzel.
- Es darf keine Kreise in der Struktur geben
- Baum heisst binärer Suchbaum, falls für jeden Knoten v gilt. Der `key` von jedem Knoten im **linken** Teilbaum `v->left` ist kleiner/gleich `v->key`. Der `key` von jedem Knoten im **rechten** Teilbaum `v->left` ist grösser/gleich `v->key`.

6.2 Baum Suchen



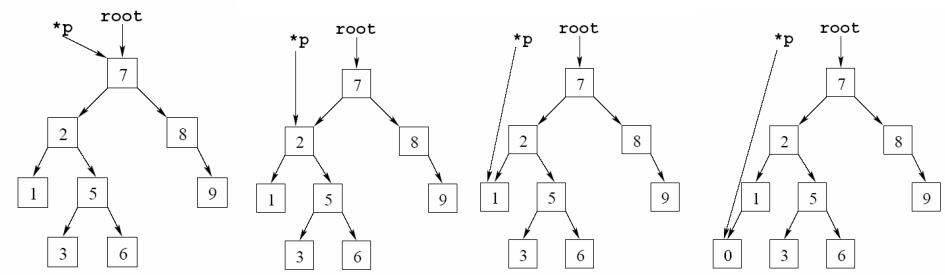
Iterativ:

```
struct element *p = root;
while(p != 0) {
if (p->key == k) return p;
if (p->key > k) p = p->left;
else p = p->right;
}
return 0;
```

Rekursiv:

```
struct element* search(struct element* p, int k) {
   if (p == 0) return 0;
   if (p->key == k) return p;
   if (p->key > k) return search(p->left,k);
   return search(p->right,k);
}
```

6.3 Baum Einfügen


```

Iterativ:

```
struct element** p = &root;
   while(*p != 0) {
   if ((*p)->key > k) p = &((*p)->left);
   else p = &((*p)->right);
   }
*p = new struct element
```

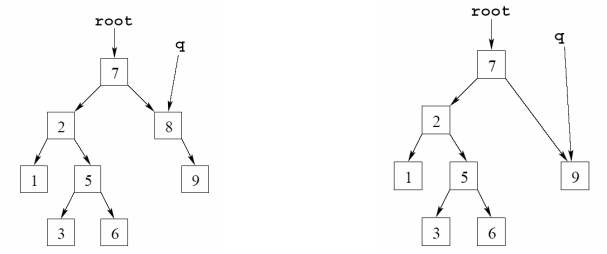
Rekursiv:

```
void insert(struct element* &p, int k) {
   if (p == 0) {
   p = new struct element;
   p->key = k;
   p->left = 0;
   p->right = 0;
   }
if (p->key > k) insert(p->left,k);
insert(p->right,k);
}
```

- Aufruf: `insert(root,k)`
- insert Funktion sucht Blatt (`if (p==0)`) und fügt neuen Pointer ein.

6.4 Baum Löschen

1.Fall: Mindestens ein Teilbaum ist leer:



Iterativ:

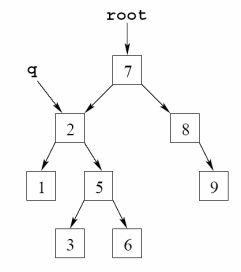
```
struct element* &q = p->right; // bzw. p->left
   if (q->right == 0 || q->left == 0) {
   struct element* h = q;
   if (q->right == 0) q = q->left;
   else q = q->right;
   delete h;
}
```

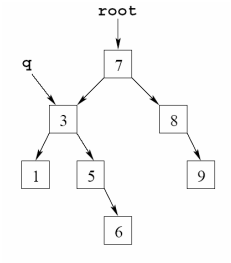
Rekursiv:

```
void remove(struct element* &q, int k) {
   if (q == 0) return; // nicht gefunden
   if (q->key == k) {
      if (q->right == 0 || q->left == 0) {
            struct element* h = q;
            if (q->right == 0) q = q->left;
            else q = q->right;
            delete h;
      }
   }
   if (q->key > k) remove(q->left,k);
   remove(q->right,k);
}
```

2.Fall: Beide Teilbäume nicht leer:

- Suche im rechten Teilbaum das Element mit dem kleinsten Schlüssel und kopiere dessen Inhalt nach *q. Lösche dann kleinstes Element im Blatt.





Rekursiv:

```
q->key = remove smallest(q->right);

int remove smallest(struct element* &p) {
   if (p->left == 0) {
      int h = p->key;
      struct element* temp = p;
      p = p->right;
      delete temp;
      return h;
   }
   return remove smallest(p->left);
}
```

6.5 Baum Ausgabe

Rekursiv:

```
void print(struct element* p) {
   if (P==0) return;
   print(p->left);
   cout << p->key;
   print(p->right);
}
```

7 Beispielprogramme

7.1 Arrays suchen

Iterativ:

```
int LineareSuche(int a[N], int b) {
   for (int i = 0; i < N; i++)
      if (a[i] == b) return i;
   return -1;
}

int BinaereSuche(int a[N], int b) {
   int left = 0, middle, right = N-1;
   while (left <= right) {
      middle = left + (right - left)/2;
      if (a[middle] == b) return middle;
      else
        if (a[middle] < b)
          left = middle + 1;
        else right = middle - 1;
   }
return -1;
}
```

Rekursiv:

```
int BinaereSuche(int* a, int b, int N) {
   if (N <= 0) return -1; // nicht gefunden
   int middle = N / 2
   if (a[middle] == b) return middle;
   else
      if(a[middle] < b)
         return middle + 1 +
            BinaereSuche(a+(middle+1),b,N-(middle+1));
      else return BinaereSuche(a,b,middle);
}
```

Teilstringsuche:

```
int main(){
.
.
.
text_N=strlen(text);
pattern_N=strlen(pattern);

for(i=0;i<text_N;i++){
        if(pattern[0]==text[i]){
                for(j=1;j<pattern_N;j++){
                        if(pattern[j]!=text[i+j]) break;
                        if(j==pattern_N-1) {
                        cout << "Pattern beginnt an " << i+1 << ". Stelle";
                        return 0;
                        }
                }
        }
}
cout << "Pattern ist nicht im Text vorhanden";

}
```

7.2 Arrays sortieren

7.2.1 Sortieren durch Maximumssuche

```
int Maximum(int a[N], int N) {
   int max = a[0], index = 0;
   for (int i = 1; i < N; i++)
      if (a[i] > max) { max = a[i]; index = i; }
   return index;
}
void SortMax(int a[N], int N) {
   if (N == 1) return;
   int i = Maximum(a,N);
   int h = a[i];
   a[i] = a[N-1];
   a[N-1] = h;
   SortMax(a,N-1);
}
```

7.2.2 Quicksort

```
void swap(int& a, int& b)
{
   int temp=b;
   b=a;
   a=temp;
} // swap the value of two variables

void sort(int array[], int begin, int end) {
   if (end - begin > 1) { // if subarray is too small, sorting done
      int pivot = array[begin]; // select pivot
      int l = begin + 1;
      int r = end;
      while(l < r) {
         if (array[l] <= pivot) {
            l++;
         } else {
            r--;
            swap(array[l], array[r]);
         }
      }// sort wrt pivot
      l--;
      swap(array[begin], array[l]);
      sort(array, begin, l); // proceed iteratively
      sort(array, r, end);
   }
}
int main(){
   sort(a,0,8);         //Array mit 8 Elementen
}
```

7.2.3 Bubblesort

```
void bubble_step(int a[], int end)
{
   int temp;
   for(int i=0; i<end; i++)
   {
   if(a[i]>a[i+1]){
      temp = a[i];
      a[i] = a[i+1];
      a[i+1] = temp;
   }
   }
}
void bubblesort(int a[], int N)
{
   for(int i=N-1; i> 0; i--) bubble_step(a, i);
}
```

7.3 Rekursion

7.3.1 Rencontre

```
int rencontre(int n)
```

```
{
   int r[n+1];
   r[0]=1;
   if(n>0) r[1]=0;
   if(n>=2) {
   for(int i=2; i<=n; i++)
      r[i] = (i-1) * (r[i-1] + r[i-2]);
   }
   return r[n];
}
```

7.3.2 Fibonacci

```
#include<iostream>
using namespace std;

char tab= '\t';
int i;

void forward(int n, int depth){
       for(i=0;i<depth;i++){cout << tab;}
       if(depth==0){cout << "fib("<< n << ")"<< endl;}
       else{cout << "--> fib("<< n << ")"<< endl;}
}

void backward(int n,int back, int depth){
       for(i=0;i<depth;i++){cout << tab;}
       if(depth==0){cout << "Endergebnis:"<< back<< endl;}
       else{cout << "<--"<< back<< endl;}
}

int fib(int n, int depth){
       int back; //wert von fib in gegebener Tiefe depth

       forward(n,depth);
       if(n==0) back=0;
       if(n==1) back=1;
       if(n>1) back=fib(n-1,depth+1)+fib(n-2,depth+1);
       backward(n,back,depth);
       return back;
}

int main(){

int n;
int depth=0;

cout << "Fibonacci-Folge fuer wievieltes Glied? ";
cin >> n;
fib(n,depth);

}
```

7.4 Tracing

nach	globale Variablen					p1		p2		p3	
	a	b	c	d[0]	d[1]	x	y	a	b	u[0]	u[1]
Zeile 31	1	3	5	3	4						
Zeile 33,5						=a	3				
Zeile 10	6	3				6	3				
Zeile 34,13								6	=b		
Zeile 17		9	13					4	9		
Zeile 35,20										3	4
Zeile 22,13								3	=u[1]		
Zeile 17			13		12			1	12	3	12
Zeile 23,5						=u[0]	12				
Zeile 10			12	8		8	4			8	
Zeile 36	6	9	12	8	12						

7.5 Vertauschen (Swap)

Referenzen:

```
void swap2(int &a, int &b)
{
   // a temporary variable for swapping.
   int temp;
   temp = a;
   a = b;
   b = temp;
}
// calling the swap function.
// no need to use '&' since the parameters
// are passed by reference.
swap2(x, y);
```

Pointer:

```
void swap3(int *a, int *b)
{
   // a temporary variable for swapping.
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}

// calling the swap function.
// we call using the addresses of x, y.
swap3(&x, &y);
```

7.6 Aymptotik

Aufgabe 1: Asymptotik (theoretische Aufgabe)

Gegeben zwei Funktionen $f : \mathbb{N}_+ \to \mathbb{N}_+$, $g : \mathbb{N}_+ \to \mathbb{N}_+$. Man sagt, f *wächst asymptotisch mindestens so schnell* wie g, falls es ein $c > 0$ gibt, so dass

$$g(n) \leq cf(n) \quad \text{für alle } n.$$

In diesem Fall, schreibt man $g \leq \mathcal{O}(f)$. Falls es kein solches c gibt, sagt man, g *wächst asymptotisch schneller* als f.
Wir definieren $f_i : \mathbb{N}_+ \to \mathbb{N}_+$, $1 \leq i \leq 4$, wie folgt:

$$\begin{array}{ll} f_1(n) = n^3, & f_2(n) = 7n^2, \\ f_3(n) = 5n^2 + n, & f_4(n) = 5n^2 \end{array}$$

Bestimmen Sie, welche der folgenden Aussage richtig sind und begründen Sie Ihre Antwort.

a) $f_4 \leq \mathcal{O}(f_2)$ b) $f_2 \leq \mathcal{O}(f_4)$ c) $f_1 \leq \mathcal{O}(f_2)$
d) $f_2 \leq \mathcal{O}(f_3)$ e) $f_3 \leq \mathcal{O}(f_2)$ f) $f_3 \leq \mathcal{O}(f_4)$

a) **true**. Set $c = 1$.

b) **true**. Set $c = 2$.

c) **false**. Since $\lim_{n\to\infty} f_1(n)/(c \cdot f_2(n)) = \infty$ for every $c > 0$, there is some $n_0 \in \mathbb{N}$ (depending on c), such that $f_1(n) > cf_2(n)$, for $n > n_0$.

d) **true**. Set $c = 2$.

e) **true**. $f_3(n) = 5n^2 + n \leq 5n^2 + n^2 \leq 1 \cdot 7n^2 = f_2(n)$.

f) **true**. $f_3(n) = 5n^2 + n \leq 6n^2 \leq 2 \cdot 5n^2 = 2f_4(n)$.