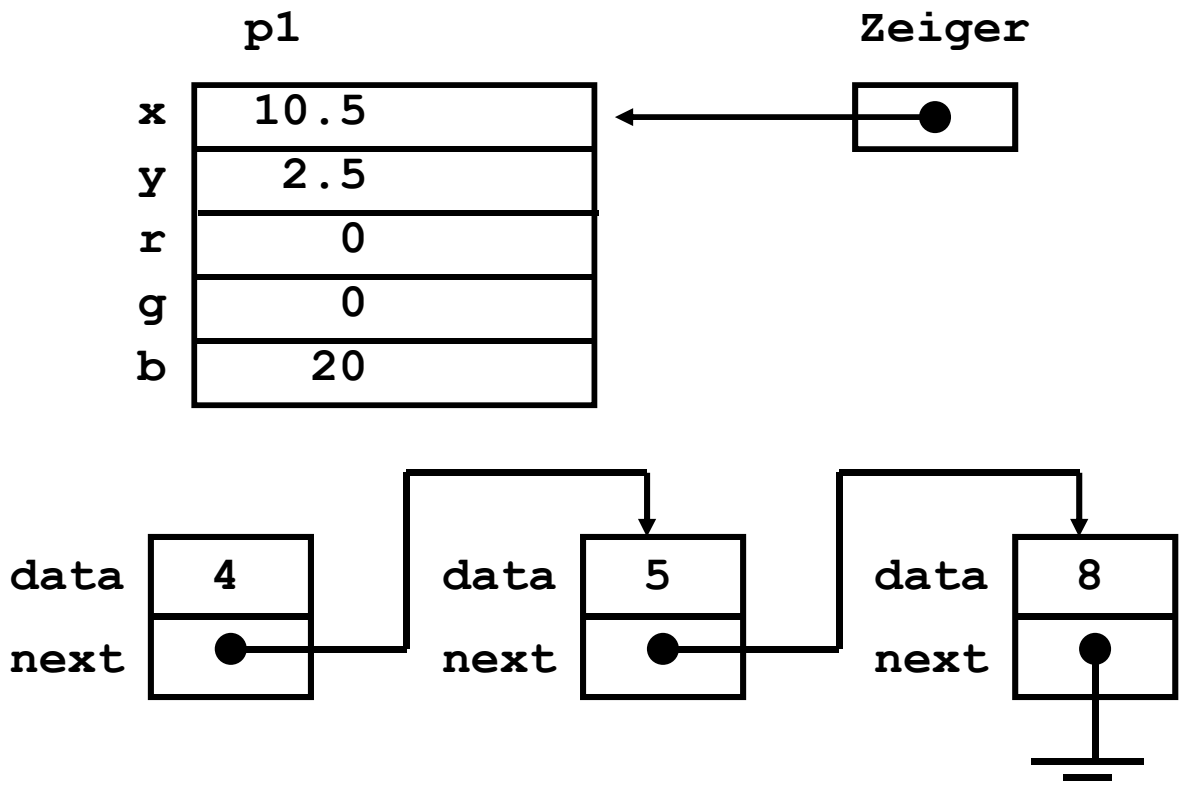
Modulbezogene Übung C++ zur Vorlesung Computer Graphics

Prof. Dr. Martin Hering-Bertram
Hochschule Bremen



Literatur:

Ulrich Breymann: C++: eine Einführung, Hanser Verlag, 2016

Inhalt

0. Erste Beispiele in C++

Von Java nach C++

C wurde entwickelt, um maschinennahen Code zu programmieren, welcher möglichst genauso effizient ist, wie in Assembler implementierte Maschinensprache. Der Preis für Effizienz sind Kompromisse beim Komfort. So sind sämtliche Variablen und Funktionen vor ihrer Verwendung zu **deklarieren**, denn der Compiler "schaut" nicht nach vorne. Die Gültigkeit von **Zeigern** (Referenzen auf Elemente im Speicher) wird nicht überprüft, was zu häufigen Programmabstürzen und zu nervenraubendem Debugging führen kann. Weiterhin gibt es keinen **Garbage Collector**, der hinter dem Programmierer "herläuft und aufräumt".

C++ ist die objektorientierte Erweiterung und unterstützt den vollen Funktionsumfang von C. Hier gibt es etwas mehr Komfort durch die Einführung von **Klassen** und **Templates**, aber man benötigt im Vergleich zu Java immer noch deutlich mehr Wissen und Disziplin, um "sauber" zu programmieren.

0. Erste Beispiele in C++

Module und Schnittstellen

Softwareprojekte enthalten in der Regel mehrere Module, bestehend aus einer *.c bzw. *.cpp Datei mit der eigentlichen Implementierung und einer **Headerdatei** *.h, welche das Interface zu anderen Modulen darstellt. Das Compilieren und Linken (Zusammenfügen) der Module wird über ein **Makefile** gesteuert, welches allerdings von den meisten Entwicklungsumgebungen im Hintergrund verwaltet wird. Wurde nur ein Dokument editiert, so **übersetzt** der **Compiler** nur die Module mit Abhängigkeiten neu und der **Linker** fügt die vorhandenen und neu entstandenen **Objektdateien** *.o zu einem ausführbaren Programm (**Executable**) zusammen.

Bevor wir tiefer in die Programmierung einsteigen, betrachten wir das Beispiel **PriorityQueue.cpp** . Es handelt sich um eine Datenstruktur, die Ganzzahlen aufnimmt und durch Abfrage des kleinsten Elementes in sortierter Reihenfolge ausgibt. Der Einfachheit wurde auf eine Headerdatei verzichtet.

0. Erste Beispiele in C++

Priority Ququeue

Das Modul beginnt mit einem Kommentar zum Inhalt:

```
// Priority Queue Example
// Author: MHB
// Date: 10.2017
```

Einzeilige Kommentare können mit **//** begonnen werden, für Mehrzeilige empfiehlt sich **/*** ... ***/** .
Es folgen die mit **#** eingeleiteten Compileranweisungen

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

um die verwendeten Bibliotheken für Ein/Ausgabe, das Template für die Datenstruktur **Queue** und **math.h** für mathematische Funktionen zu verwenden.

0. Erste Beispiele in C++

```
using namespace std;
```

ermöglicht es, auf den Zusatz **`std::`** z.B. bei Verwendung von **`std::cin`** und **`std::cout`** zu verzichten.

Das Hauptprogramm besteht immer aus der Funktion

```
int main(){
```

welche optional einen ganzzahligen Wert über den Erfolg und Misserfolg des Programms zurückgibt.

Als nächstes definieren wir ein **Array** fester Größe. Der Speicherbereich dafür wird nur einmal **statisch allokiert** und ist nicht in seiner Größe variabel:

```
const int n=6;
int numbers[n] = {1, 5, 8, 2, 3, 2};
```

0. Erste Beispiele in C++

Der **Wertebereich** für die Indizierung des Arrays **`numbers[i]`** ist 0, .., 5. Würde man z.B. auf **`numbers[6]`** oder **`numbers[-1]`** zugreifen, führt dies nicht zu einem Fehler. Es ist aber möglich, dass das Programm abstürzt, wenn man durch unkontrollierten Zugriff andere Datenstrukturen, die für den Ablauf des Programms sorgen, überschreibt.

Da wir hier mit **Ganzzahlen** (**`int`**) arbeiten, ist zu bemerken, dass arithmetische Operationen auf diesem Datentyp bei Bedarf abgerundet werden. So liefert der Term **`14/5`** den Wert 2, und nicht etwa 2.8. Für letzteres müsste mindestens ein Operand als **`float`** oder **`double`** vorliegen, z.B. **`14.0/5`** .

Achtung, Fehler!

Berechnungen mit Ganzzahlen wo Gleitkommazahlen gebraucht würden, sind ein "beliebter" Fehler von Programmieranfängern in C und C++.

0. Erste Beispiele in C++

Nun folgt erstmalig die Verwendung eines **Templates**: einer Datenstruktur, die für beliebige Datentypen einsetzbar ist: Die PriorityQueue **`Q`** erhält **`int`** als Datentyp für die enthaltenen Elemente, **`vector<int>`** (ein weiteres Template) für die interne Realisierung, sowie einen Vergleichsoperator "<" für Ganzzahlen. Für den Gebrauch dieser Templates ist die o.g. Include-Anweisung notwendig.

```
priority_queue< int, vector<int>,
     less<int> > Q;
```

Als nächstes wollen wir die Schlange mit den gegebenen Zahlen füllen und geben folgende Ankündigung aus:

```
cout << "The input list is:" << endl;
```

Mehrere Werte und Zeichenketten können, durch den Operator **`<<`** getrennt, durch **`cout`** verarbeitet werden. **`endl`** steht für den Zeilenumbruch.

In C würde man die Funktion **`printf()`** aus der Bibliothek **`stdio.h`** verwenden.

0. Erste Beispiele in C++

Die **for-Schleife**

```
for( int i=0; i<n; i++){
```

enthält eine **Initialisierung** (hier mit Deklaration der Variablen `i`), einen **logischen Ausdruck** (Vergleich), welcher über die Fortsetzung entscheidet, und eine **Zählanweisung**, die hier den Wert von `i` inkrementiert. Die Variable `i` existiert nur innerhalb des Schleifenkörpers. Braucht man den Wert noch nach Ablauf der Schleife, so müsste `i` weiter oben im Programm deklariert werden.

```
    cout << numbers[i] << endl;
    Q.push( numbers[i]);
}
cout << endl;
```

Hier wird die i-te Zahl des Arrays ausgegeben und in die PriorityQueue eingefügt. Es empfiehlt sich die Ausgabe eines Zeilenumbruchs am Ende der Schleife, damit alle Werte ausgegeben werden und nicht in einem **Puffer** "verenden".

0. Erste Beispiele in C++

Zuletzt erfolgt die Ausgabe der sortierten Liste, indem immer die kleinste Zahl aus der PriorityQueue entnommen und ausgegeben wird:

```
cout << "The sorted list is:" << endl;
while( Q.size() > 0){
      cout << Q.top() << endl;
      Q.pop();
}
```

Die **while-Schleife** wird solange durchlaufen, wie noch Elemente in der PriorityQueue vorhanden sind. Mit `top()` liest man das "oberste" (kleinste) Element; mit `pop()` wird es gelöscht. Die Sortierung wird von der Datenstruktur eigenständig vorgenommen. hierbei wird der in der Deklatation festgelegte Vergleichsoperator aufgerufen. Man kann solche Operatoren **überladen**, d.h. neu implementieren, derart dass die Arithmetik einer eigenen Logik folgt.

0. Erste Beispiele in C++

Klammer Vergessen?

Beim Aufruf von Methoden (oder Funktionen) ohne Argumente, wie `Q.top()`, darf man nicht die Klammern vergessen. Mit `Q.top` würde nicht unbedingt ein Fehler generiert, sondern ein Zeiger auf die Funktion zurückgegeben. Kann der Compiler den Datentyp des Zeigers nicht an `int` zuweisen, erhält man allerdings eine Fehlermeldung. Oft bleiben derartige Fehler jedoch unerkannt und bescheren dem unachtsamen Programmierer viele Überstunden.

Natürlich ist auch die Funktion `main()` zu beenden:

```
}
```

Man könnte vorher noch mit `return` einen Wert zurückgeben. Gleichermaßen kann die Funktion `main()` auch (z.B. bei Aufruf des Executables aus der Konsole) mit Argumenten versorgt werden, die im Programm abrufbar wären.

0. Erste Beispiele in C++

Priority Queue mit Brüchen

Um die Mächtigkeit von Templates zu demonstrieren, dient das Beispiel **PriorityQueueMitBruechen.cpp** . An Stelle von Ganzzahlen kann man eine eigene Klasse für den Datentyp mit Vergleichsoperator definieren und die gleiche PriorityQueue damit füttern:

```
class bruch{
public:
    int zaehler, nenner;

    bruch(); // empty constructor
    bruch( int a, int b); /* constructor
                ... with initialization */

    friend bool operator<(
        bruch x, bruch y); // less-op.
};
```

0. Erste Beispiele in C++

Priority Queue mit Brüchen

Die Klasse **`bruch`** enthält neben dem Standard-Konstruktor **`bruch()`** und dem (hier nicht deklarierten) Destruktor **`~bruch()`** noch einen Konstruktor mit Parametern zur Initialisierung.

Der Vergleichsoperator **`operator<()`** ist zwar keine Methode der Klasse, darf als "friend" aber auch auf private Elemente der Klasse zugreifen.

Die o.g. Methoden und Funktionen sind hier nur **deklariert**, d.h. es fehlt die Implementierung. Diese erfolgt im Rahmen der **Definition** später im Programm:

```
bruch::bruch(){}; // empty constructor

bruch::bruch( int a, int b){
     zaehler = a;
     nenner = b;
}
```

0. Erste Beispiele in C++

```
bool operator<( bruch x, bruch y){
    return
        x.zaehler / (float)x.nenner <
        y.zaehler / (float)y.nenner;
}
```

Ohne die Umwndlung (wenigstens eines) Operanden in **`float`** würde in Integer gerechnet und das Ergebnis wäre ggf. falsch. Trotzdem gibt es Beispiele, bei denen der Fehler nicht auftritt, z.B. **`3/4 < 5/4`** ergibt **`0<1`**.

Das Hauptprogramm ist fast identisch mit dem des ersten Beispiels, nur dass die Queue nicht mit **`int`** sondern mit **`bruch`** als Datentyp deklariert wird:

```
priority_queue< bruch, vector<bruch>,
    less<bruch> > Q;
```

0. Erste Beispiele in C++

Die Methoden der Queue sind analog des ersten Beispiels verwendbar, z.B. mit

```
Q.push( bruch( 3, 4));
cout << Q.top().zaehler;
cout << Q.top().nenner;
Q.pop();
```

Natürlich könnte man den Zugriff auch jeweils mit einem **Getter** und **Setter** realisieren. Die Bestandteile der Klasse können mit **`public:`** öffentlich und mit **`private:`** nur innerhalb der Klasse (für Methoden und Friends) zugreifbar gemacht werden.

Durch die Verwendung von Gettern und Settern kann man bei komplexeren Klassen ggf. Inkonsistenzen in der Datenstruktur durch unautorisiertes Zugreifen vermeiden, sofern die Inhalte als **`private`** deklariert sind.

0. Erste Beispiele in C++

Zusammenfassung

Das Modul **`PriorityQueue.cpp`** verwendet ein Template als Datenstruktur für die Sortierung von Ganzzahlen. Das selbe Template wird im Modul **`PriorityQueueMitBruechen.cpp`** für einen eigenen Datentyp **`bruch`** eingesetzt.

Die Deklaration und Ausgabe von Variablen und die Schleifenkonstrukte **`for`** und **`while`** (es gibt auch noch **`do-while`**) wurden exemplarisch erläutert. Einige Eigenarten der Programmiersprachen C und C++ mit möglichen Fehlerquellen wurden diskutiert.

Einen Überblick über C++ Templates findet man hier:

https://de.wikipedia.org/wiki/Template_(C%2B%2B)
https://de.wikipedia.org/wiki/C%2B%2B-Standardbibliothek
https://www.cplusplus.com/reference/
http://stdcxx.apache.org/

1. Grundlagen

Programmieren und Testen

1. Eingeben der Sourcen mit einer Entwicklungsumgebung (IDE, z.B. Code::Blocks, Visual Studio, QT) oder einem Editor (z.B. **`vi`**)
2. Compilieren des Programms (Umsetzung in Maschinensprache), z.B.

   `gcc` *`name`***`.c -ansi`** (für ANSI-C)

   `g++` *`name`***`.cpp`** (für C++)

   (generiert **`a.out`**)
3. Ausführen und Testen, z.B.

   `a.out`

   (ruft das Maschinenprogramm auf)

Bei Verwendung mehrerer Module müssen diese vor dem Ausführen durch **Linken** zusammengefügt werden. Dies wird meistens durch ein **`Makefile`** geregelt (vgl. **`make`**). Bei Verwendung einer IDE wird dies mit "build & run" automatisch ausgeführt.

1. Grundlagen

Compiler-Optionen

`-g` : Debugger Unterstützung beim Compilieren. Der Debugger (z.B. **`gdb`**) liefert Statusinformationen nach einem Programmabsturz und dient zum Auffinden von Programmierfehlern.

`-O` : Optimieren des Maschinenprogramms. Das Ausführen des Programms wird oft schneller, jedoch dauert das Compilieren länger. Diese Option hat in Kombination mit **`-g`** keine Wirkung.

`-o` *name* : Generiert ein Maschinenprogramm mit vorgegebenem Namen (statt **`a.out`**)

`-lm` : Linkt die Mathematik Bibliothek hinzu. Diese liefert z.B. trigonometrische Funktionen.

Beispiel: **`g++ hello.cpp –g –o hello -lm`**
Weitere Informationen z.B. unter Linux mit **`man g++`**

1. Grundlagen

Von C nach C++: Hello-World Programm

(gibt **`hello, world`** im Textfenster aus):

```
/* in C */
#include <stdio.h>
int main()
{
    printf("hello, world\n");
}

// in C++
#include <iostream>
int main()
{
    std::cout << "hello, world\n";
}
```

1. Grundlagen

Erläuterung

`#include <stdio.h>`

ist eine Preprozessor Anweisung und veranlasst den Compiler, die Standard I/O Bibliothek (**`stdio`**) zu verwenden. (z.B. für die Funktion **`printf`**)

`int main()`

ist die Erste **Funktion**, die bei Programmstart aufgerufen wird. Nach Abarbeiten dieser Funktion terminiert das Programm

`{` Klammerung umschließt den Funktionsrumpf

`printf ("hello, world\n");`
Textausgabe (**`\n`** steht für „Neue Zeile“)
(Semikolon nach jeder Anweisung)

`}` Programmende

1. Grundlagen

Kommentare

Kommentare zur Dokumentation eines Programms beginnen mit **/*** und enden mit ***/** (Verschachtelung ist nicht erlaubt). Kommentare werden vom Compiler ignoriert. Nur in C++: **//** Kommentar bis zum Zeilenende

Programme sollten immer gut dokumentiert und leserlich formatiert werden.

Beispiel:

```
int main(){ // Hauptprogramm

    Hello(); // schreibt "hello"

    // die naechste Funktion
    // schreibt "world"
    World();
}
```

1. Grundlagen

Funktionen

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

void Hello(){
    cout << hello" << endl;
}

void World(){
    cout << ("world" << endl;
}

int main(){
    Hello(); World(); Hello();
}
```

Liefert die Ausgabe:

```
hello
world
hello
```

1. Grundlagen

Funktionen

Funktionen eignen sich für Programmteile die semantisch zusammengehören und mehrfach benötigt werden (Modularisierung des Programms).

Das Hauptprogramm wird dadurch sehr kurz und übersichtlich.

Funktionen können geschachtelt aufgerufen werden. **Die aufgerufene Funktion muss vor der aufrufenden Funktion deklariert sein, da der Compiler sie sonst noch nicht kennt**. Beispiel:

```
void A(){ /* wird von B aufgerufen */
     ...
}

void B(){ /* ruft A auf */
     A();
}
```

1. Grundlagen

Funktionen

Die Reihenfolge der Funktionen darf vertauscht werden, wenn diese vorher **deklariert** werden. Die Deklaration enthält keinen Funktionsrumpf:

```
void A(); /* deklaration */
void B(); /* deklaration */

void B(){ /* ruft A auf */
    /* A ist dem Compiler bekannt, da
       es oben deklariert wurde */
    A();
}

void A(){ /* wird von B aufgerufen */
    ...
}
```

1. Grundlagen

Variablen

Daten wie z.B. Ganzzahlen (**`int`**), Fließkommazahlen (**`float`**), oder alphanumerische Zeichen (**`char`**) werden in Variablen gespeichert.

Lokale Variablen werden am Funktionsanfang deklariert und gelten nur innerhalb der Funktion.

In C ist erfolgt die Ausgabe nach Datentypen, was in C++ durch **`cout << ...`** automatisch geschieht:

```
int main(){
    float    f = 3.5;
    int      i = 5;
    char     c = 'A'

    /* nicht vergessen: stdio.h */
    printf("%f\n", f); /* %f float */
    printf("%d\n", i); /* %d int */
    printf("%c\n", c); /* %c char */
}
```

1. Grundlagen

Variablen

Die Deklaration lokaler Variablen ist notwendig, um den **Variablentyp** zu vereinbaren. Das Zuweisen eines Wertes ist optional:

```
float    f;
```

und

```
float    f = 3.5;
```

sind beides gültige Deklarationen.

Bei der Ausgabe von Zahlen mit **`printf`** ist eine **Format-Zeichenkette** gefolgt von den zugehörigen Werten erforderlich.

```
printf("Ergebnis: %f %d\n", 3.2, 3.2);
```

liefert z.B. die Ausgabe

```
Ergebnis:       3.2       3
```

hierbei wurde der zweite Wert als Ganzzahl interpretiert und daher abgerundet.

1. Grundlagen

Variablen

In C++ kann man sich aussuchen, ob man die "alte" C-Schreibweise verwendet, oder mit der Standard-bibliothek **`iostream`** arbeitet. Empfehlenswert ist letzteres. Hier stehen folgende Möglichkeiten zur formatierten Ausgabe bereit:

```
cout.width(10);     // insg. 10 Stellen
cout.precision(4); // 4 Nachkommastellen
cout << hex; // als Hexadezimalzahl
cout << dec; // als Dezimalzahl
```

dies bezieht sich auf die nachfolgenden Ausgaben. Bei

```
cout << f;
```

wird der Datentyp automatisch erkannt.

1. Grundlagen

Variablen

Die Zuweisung von Werten kann und sollte innerhalb des Funktionsrumpfes geschehen:

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

    f = 4.8;
    i = f;
    j = -f;

    cout << i << " " << j << endl;
}
```

Liefert die Ausgabe

```
    4   -4
```

(es wird immer in Richtung der Null gerundet).

1. Grundlagen

Ausdrücke

Der Operator **=** kann die Werte beliebiger Ausdrücke (rechts) einer Variablen (links) zuweisen. Die Reihenfolge der Auswertung eines Ausdrucks erfolgt von links nach rechts, wobei Punkt vor Strich gilt. Die vier Grundrechenarten sind durch die Symbole **+**, **-**, ***** und **/** definiert:

```
int    i, j;
float  f, g;
i = 2 + 3 * 3;        /* i = 11 */
j = (2 + 3) * 3;      /* j = 15 */
f = i / 2.0 + 0.5;    /* f = 6.0 */
g = i / 2 + 0.5;      /* g = 5.5 */
```

Im letzten Beispiel wird die Division in Ganzzahl-Arithmetik durchgeführt, da beide Operanden vom Typ **int** sind.

1. Grundlagen

Ausdrücke

Als Operanden sind nicht nur Variablen und Konstanten erlaubt, sondern auch Funktionen, welche einen Wert zurückliefern. Beispiel:

```
float Quadrat( float f){
    float g = f * f;
    return g;
}
```

Diese Funktion hat ein Argument **`f`** vom Typ **`float`** und gibt einen Wert vom Typ **`float`** zurück. Letzteres geschieht mit der **`return`** Anweisung. Die Funktion kann nun als Operand in einem Ausdruck aufgerufen werden, z.B.

```
h = 2.0 + Quadrat( 0.5); /* h = 2.25 */
```

Die Lokalen Variablen **`g`** und **`f`** sind nur innerhalb der Funktion **`Quadrat`** definiert. Gleichnamige Variablen in einer aufrufenden Funktion sind davon völlig unabhängig.

1. Grundlagen

Bezeichner

Variablen- und Funktionsnamen sind **Bezeichner**, d.h. vom Programmierer gewählte Namen. Diese bestehen aus einer zusammenhängenden Folge alpha-numerischer Symbole, wobei das erste Zeichen ein Buchstabe sein muss. Beim Aneinanderreihen mehrerer Wörter kann das Zeichen '_' oder Groß- / Kleinschreibung verwendet werden. Es gibt jedoch bestimmte Schlüsselwörter, die nicht als Bezeichner eingesetzt werden dürfen, wie z.B.

auto	**break**	**case**	**char**
const	**continue**	**default**	**do**
double	**else**	**enum**	**extern**
float	**for**	**goto**	**if**
int	**long**	**register**	**return**
short	**signed**	**sizeof**	**static**
struct	**switch**	**typedef**	**union**
unsigned	**void**	**volatile**	**while**

in C++ auch noch **class, delete, new, ...**

1. Grundlagen

Bezeichner

Beispiele für gültige Bezeichner sind:
`Zaehler, Zaehler1, i, i23, i2_j,`
`register1, Register, XYZ, A, B,`
`AB, int_funktion, intFunktion`

Als Bezeichner **ungeeignet** sind z.B.
`int, register, 2i, i-23, A*B`

Bezeichner besitzen in C 32 signifikante Zeichen (in C++ fast beliebig viele), dürfen aber länger sein.

1. Grundlagen

Eingabe

Die **stdio.h** stellt in C auch Funktionen für die Eingabe von Zahlen und Zeichenketten bereit, z.B. **scanf** :

```
int    i;
float  f;
scanf("%d", &i);
scanf("%f", &f);
```

Dieses Programmsegment fordert den Benutzer auf, eine Ganzzahl und eine Fließkommazahl einzugeben, deren Werte den Variablen **i** und **f** zugewiesen werden.

Anders als bei **printf** muss hier die **Adresse** der Variablen als Argument übergeben werden, damit deren Wert überschrieben werden kann. Die Adresse einer Variablen liefert der Operator **&**.

1. Grundlagen

Eingabe

In C++ verwendet man vorzugsweise **cin**:

```
char   c;
int    i;
float  f;
string s; // mit #include<string>
cin >> c;
cin >> i;
cin >> f;
cin >> s;
cout << c << i << f << s << endl;
```

Hier werden die Eingaben typgerecht zugewiesen. In C müssen Zeichenketten als Arrays von char verwaltet werden, z.B.

```
char s[1024];  /* s=Zeiger auf char[0] */
scanf("%s", s);/* ohne & */
```

2. Iteration und Rekursion

Iteration

Iterationen (wiederholte Ausführungen eines Blocks) können durch **`for`**-Schleifen realisiert werden:

```
for (Init; Bedingung; Zähl){...}
```

Beispiel:

```
int i;
for (i=0; i<10; i++){
    ... /* wird 10 mal ausgefuehrt */
}
```

Die **`for`**-Anweisung enthält drei Ausdrücke:

-Eine **Initialisierung**, hier **`i=0`**, welche vor Beginn der Iteration ausgeführt wird.

-Eine **Bedingung**, hier **`i<10`**, welche erfüllt sein muss damit die Schleife erneut durchlaufen wird.

-Eine **Zählanweisung**, hier **`i++`** (analog zu **`i=i+1`**), welche nach jedem Durchlauf ausgewertet wird.

2. Iteration und Rekursion

Iteration

Beispiel: Berechnung der Fakultät

```
/* Fakultaet von n */
int fact( int n){

    int i, Ergebnis = 1;

    /* Iteration zur Berechnung von n! */
    for (i=1; i<=n; i++){
        Ergebnis = Ergebnis * i;
    }

    return Ergebnis;
}
```

Hinweis: eine gleichnamige Funktion für die Fakultät befindet sich auch in der **`math.h`**.

(**`#include <math.h>`** , mit der Compileroption **`-lm`**)

2. Iteration und Rekursion

Iteration

Beispiele für Fehler:

```
for (i=0; i>10; i++){
    /* Block wird nie ausgefuehrt */
    ...
}

for (i=0; i<10; i++);{
    /* wird nur einmal ausgefuehrt */
   ...
}

for (i=0; i>=0; i++){
    /* Endlosschleife,
       Abbruch mit STRG-Z */
    ...
}
```

2. Iteration und Rekursion

Iterationskonstrukte

Neben der **`for`**-Schleife gibt es noch zwei andere Schleifenkonstrukte, **`while`** und **`do-while`**:

```
while ( Bedingung ){ Block }
```

```
do { Block } while ( Bedingung );
```

Die **`while`**-Schleife wertet zunächst die *`Bedingung`* aus. Bei positivem Ergebnis wird der *`Block`* abgearbeitet. Dann wird die Schleife erneut ausgewertet. Die **`while`**-Schleife terminiert also erst, wenn die Bedingung nicht erfüllt ist.

Bei der **`do-while`**-Schleife wird zuerst der *`Block`* abgearbeitet und dann erst die *`Bedingung`* ausgewertet. Bei positivem Ergebnis wiederholt sich die Ausführung der **`do-while`**-Schleife.

2. Iteration und Rekursion

Verzweigung

Die **if**–Anweisung realisiert die bedingte Ausführung eines Programmteils:

```
if (Bedingung){
     ... /* auszufuehren, falls die
              Bedingung erfuellt ist */
}
else{
     ... /* falls nicht erfuellt */
}
```

Die **else** -Anweisung ist optional.

Beispiel:

```
if (i>0){
     cout << i << " ist positiv.\n";
}
```

2. Iteration und Rekursion

Vergleichsoperatoren

Für den Vergleich zweier Werte innerhalb einer Bedingung stehen folgende Operatoren zur Verfügung:

- `<` (kleiner als)
- `<=` (kleiner oder gleich)
- `>` (größer als)
- `>=` (größer oder gleich)
- `==` (gleich)
- `!=` (ungleich)

Achtung Fehler:
Der Operator `=` ist kein Vergleichsoperator. Würde er versehentlich statt `==` in einer Bedingung eingesetzt, z.B. `if(i=0)` , so würde `i` den Wert 0 zugewiesen bekommen und die Bedingung wäre hier nicht erfüllt (der Wert von `i` würde logisch interpretiert: 0 als **falsch** und jeder andere Wert als **wahr**).

2. Iteration und Rekursion

Rekursion

An Stelle einer Iteration, z.B. für die Fakultät:

$$n! \;=\; \prod_{i=1}^{n} i \;=\; 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$$

verwendet man oft elegantere, rekursive Algorithmen:

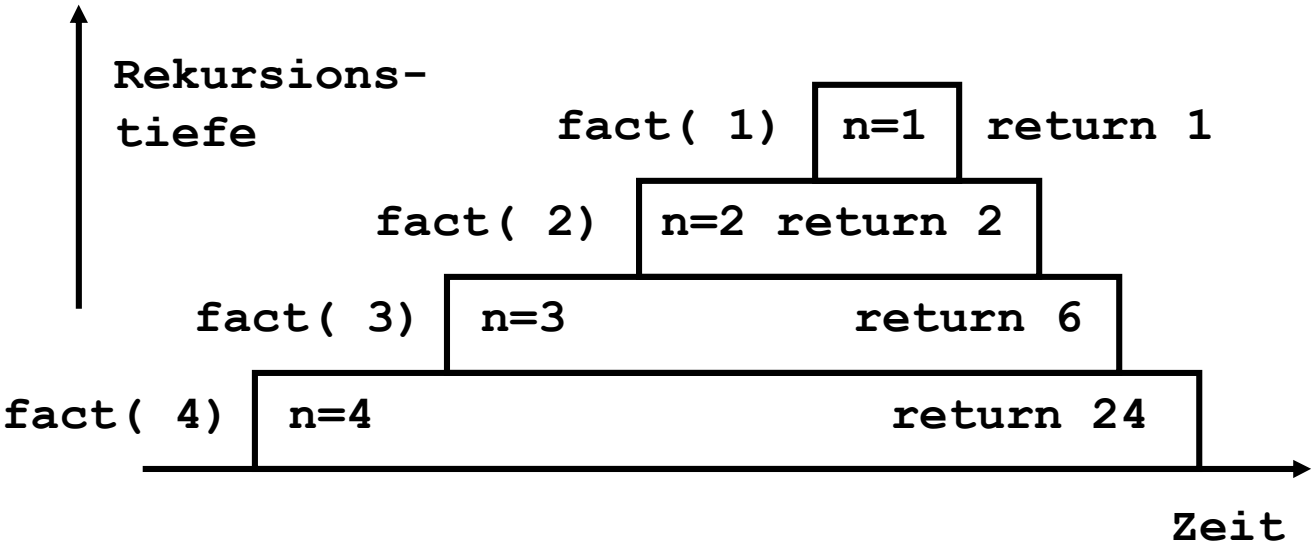
$$n! \;=\; \begin{cases} 1 & \text{falls } n \leq 1, \\ n \cdot (n-1)! & \text{sonst.} \end{cases}$$

```
int fact( int n){ /* Fakultaet von n */

    if (n <= 1){
        return 1; /* Rekursionsabbruch */
    } else {
        return n * fact( n - 1 );
    }
}
```

2. Iteration und Rekursion

Rekursion

Rekursionsaufrufe der Funktion **`fact( 4)`**



Die Rekursion benötigt mehr Speicher als die Iteration, da bei jedem Funktionsaufruf sämtliche Variablen neu angelegt werden müssen. Dennoch sind Rekursive Implementierungen meistens nicht weniger effizient (außer bei Mehrfachaufrufen oder bei sehr großen Rekursionstiefen aufgrund der Speicherbelegung).

3. Datentypen und Operatoren

Datentypen

Es gibt in C unter anderem folgende elementare Datentypen:

```
char     (Zeichen,  1 Byte)
int      (Ganzzahl, 4 Bytes)
float    (reell,    4 Bytes)
double   (reell,    8 Bytes)
void     (leerer Wertebereich)
```

Hinweis: die interne Größe eines Datentyps in Bytes ist Rechnerabhängig. Sie kann mit der Operation

```
std:size_t var = sizeof( Datentyp )
```

abgefragt werden.

3. Datentypen und Operatoren

Datentypen

Zeichen (Typ **`char`**) werden intern als Zahlen mit dem Wertebereich **–128...127** gespeichert. Daher kann man Zeichen auch vergleichen oder als Argumente arithmetischer Operationen verwenden. Dabei muss jedoch darauf geachtet werden, dass der Wertebereich nicht überschritten wird.

Gibt man den Wert eines Zeichens als Ganzzahl aus, also z.B.

```
char c='A';
printf("%d\n", (int)c);
```

so erhält man den ASCII-Wert (American Standard Code II) dieses Zeichens. Die ASCII-Werte von 0 bis 31 definieren Steuerzeichen (z.B. Newline: **`'\n'=10`**) und die Werte von 32 bis 126 definieren Sonderzeichen, Ziffern und Buchstaben.

Beispiel:

```
' '=32, '@'=64, '\n'=10,
'0'=48, '1'=49, ..., '9'=57,
'A'=65, 'B'=66, ..., 'Z'=90,
'a'=97, 'b'=98, ..., 'z'=122,
```

3. Datentypen und Operatoren

Datentypen

Um negative Werte zu vermeiden, kann man den erweiterten Typ **`unsigned char`** mit Wertebereich **0...255** verwenden. Diese Erweiterung ist auch für Ganzzahlen möglich.

Ganzzahlen werden intern als Binärzahlen dargestellt. Bei einer Größe von 4 Bytes ergibt sich daher ein Wertebereich von $\mathbf{-2^{31}...2^{31}-1}$ für den Typ **`int`** und $\mathbf{0...2^{32}-1}$ für **`unsigned int`**.

Mit den Erweiterungen **`short`** und **`long`** kann man die Größe für den Typ **`int`** verändern, jedoch ist der resultierende Typ Rechner-abhängig und nicht immer verschieden vom Basistyp. Es gibt z.B. folgende Erweiterungen:

```
unsigned char   (1 Byte)
unsigned int    (4 Bytes)
short int       (2 Bytes)
long int        (>= 4 Bytes)
long double     (>= 8 Bytes)
```

3. Datentypen und Operatoren

Datentypen

Fließkommazahlen bestehen intern aus Vorzeichenbit **s** $\in$ {-1,1}, Mantisse **m** $\in$ [0,1) und Exponent **e**. Der Wert berechnet sich dann zu $\mathbf{s \cdot m \cdot 2^e}$.

Auf diese Weise können auch sehr kleine oder sehr große Zahlen dargestellt werden. Dennoch kommt es häufig zu Rundungsfehlern. Beispiel

```
float f = 1000000;
float g = 0.0000001;
float h = f + g;
```

Der Wert von **h** ist in diesem Beispiel **h = 1000000**, da die Genauigkeit von Float für das Ergebnis nicht ausreicht. Dennoch geht der Wertebereich bis ca. $\mathbf{\pm 10^{38}}$ für **float** und $\mathbf{\pm 10^{308}}$ für **double** .

Konstanten können auch in Exponentialdarstellung (im Zehnersystem) definiert werden, z.B.

```
f = -2.5e6;  // -2500000
g = 3.0e-4;  // 0.0003
```

3. Datentypen und Operatoren

Typumwandlung (cast)

Datentypen können wie folgt umgewandelt werden:

(*Datentyp***)***Wert*

Beispiel:

```
float f = 3.5;
float g = (int) f * (int) f; // g = 9
```

Hierbei werden die Operanden jeweils in Ganzzahlen umgewandelt. Das Ergebnis **3*3=9** wird erst durch die Zuweisung an **g** in **float** umgewandelt.

In arithmetischen Ausdrücken geschieht die Typumwandlung weitgehend automatisch. Bei ungleichen Operandentypen wird der jeweils genauere Datentyp verwendet

Ersetzt man die zweite Zeile durch

```
float g = (int)( (int) f * f); // g=10
```

so erhält man **3*3.5=10.5**, dann Rundung auf **10**.

3. Datentypen und Operatoren

Arithmetische Operatoren

Es gibt in C folgende arithmetische Operatoren (**x**, **y** stehen für die Operanden):

```
x + y     Addition
x - y     Subtraktion (binäres -)
x * y     Multiplikation
x / y     Division
x % y     Modulo (Divisionsrest) für
          positive Operanden
-x        Negation (unäres -)
++x       (x = x + 1) Inkrementiert x und
          gibt den Wert von x zurück
x++       Liefert zuerst den Wert von x
          und inkrementiert x danach
--x       Dekrementiert x, analog zu ++
x--       analog zu ++
```

3. Datentypen und Operatoren

Arithmetische Operatoren

Beispiele:

```
int    i=2, j=8, k;
float  f, g=1.5;

j = j + (i++);   // i = 3,  j = 10
k = ++i;         // i = 4,  k = 4
k = j = 5;       // j = 5,  k = 5
k = 14 % 5;      // k = 4
k = 10 % 5;      // k = 0
f = g * -g;      // f = -2.25
f++;             // f = -1.25
f = k = g = 2.5;
// g = 2.5, k = 2, f = 2.0 (k ist int)
k = (i++) * (--i); // undefiniert
```

Innerhalb eines Ausdrucks hängt die Reihenfolge in der die Operanden ausgewertet werden vom Compiler ab.

3. Datentypen und Operatoren

Logische Operatoren

Die Werte **logischer Ausdrücke** (z.B. Bedingungen) sind vom Typ **`int`** oder **`bool`** (C++), wobei nur zwei Werte tatsächlich vorkomen:

`0 = "falsch"`

`1 = "wahr"`

Jeder von Null verschiedene Wert wird auch als **`"wahr"`** interpretiert. Die Vergleichsoperatoren

`==  !=  <  <=  >  >=`

liefern als Ergebnis eine Ganzzahl, die entweder den Wert Null oder Eins besitzt.

3. Datentypen und Operatoren

Logische Operatoren

Logische Operatoren sind:

```
!x          nicht x (logische Negation)
x && y      x und y
x || y      x oder y
```

Beispiele:

```
float  f = 2.0, g = 1.9999;
int    i = 2, j = 0;
( f==g && i!=1 )          // falsch
( !(f==g) || !(i!=1) )    // wahr
( i && j )                // falsch
( j=3 )                   // wahr, j=3
( 0 && (i=9) )            /* falsch,
             i=9 wird nicht ausgewertet */
```

Achtung:

Beim Vergleichen von Gleitkommazahlen ist zu berücksichtigen, dass diese mit Rundungsfehlern behaftet sein könnten.

Bei `&&` bzw. `||` werden die Operanden von links nach rechts ausgewertet bis das Ergebnis feststeht.

3. Datentypen und Operatoren

Bit-Operationen

Jedes Bit einer Ganzzahl ist als logischer Wert interpretierbar (Bit-Masken). Durch Bit-weises Verknüpfen zweier Ganzzahlen werden logische Operationen simultan ausgeführt, z.B. mit

```
~x        Bit-weise Negation
          (Einerkomplement)
x & y     x und y (binäres &)
x ^ y     x XOR y (exklusives oder)
x | y     x oder y
```

Die Bit-Schiebeoperationen << und >> ermöglichen es, die einzelnen Bits um mehrere Stellen zu verschieben (entspricht Verdopplung bzw. Halbierung pro Stelle). Beispiel:

$$00001111_2 << 3 = 01111000_2$$
$$00101100_2 >> 2 = 00001011_2$$
$$00101100_2 >> 3 = 00000101_2$$

3. Datentypen und Operatoren

Zuweisung

Wird eine Variable manipuliert und ist dabei selbst Operand, so lässt sich diese Zuweisung durch Einsetzen eines speziellen Operators oft einfacher formulieren:

```
x += y          x = x + y
x -= y          x = x - y
x *= y          x = x * y
x /= y          x = x / y
x %= y          x = x % y
x &= y          x = x & y
x ^= y          x = x ^ y
x |= y          x = x | y
x <<= y         x = x << y
x >>= y         x = x >> y
```

Eine Zuweisung ist also ein Ausdruck, mit **Seiteneffekt**. Das Ergebnis ist der zugewiesene Wert.

3. Datentypen und Operatoren

Priorität von Operationen

Die folgende Tabelle fasst von oben nach unten die Prioritäten der C-Operatoren zusammen:

Operatoren	Assoziativität
`() [] -> .`	von links
`! ~ ++ -- (`*Typ*`) sizeof` und unäre `+ - * &`	von rechts
`* / %`	von links
`+ -`	von links
`<< >>`	von links
`< <= > >=`	von links
`== !=`	von links
`&`	von links
`|`	von links
`&&`	von links
`||`	von links
`?:`	von rechts
`= += -= *= %=` etc.	von rechts

3. Datentypen und Operatoren

Priorität von Operationen

Als Faustregel kann man sich merken: Punkt- vor Strichrechnung (gilt auch für logische Operationen, wobei && an die Stelle der Multiplikation tritt). Generell werden zuerst in- und dekrementierende, dann arithmetische, vergleichende, logische, und zuletzt zuweisende Operationen ausgeführt.

Ist die Auswertungsreihenfolge nicht offensichtlich, so sollte man dennoch Klammern verwenden um die Ausdrücke übersichtlich zu gestalten.

Die Operatoren **`[] -> .`** **`&`**(unär) und **`*`**(unär) werden in den Kapiteln 5 und 6 eingeführt.

Der Operator

`(`*`Bedingung`***`) ? x : y`**

liefert den Wert **`x`**, falls die *`Bedingung`* erfüllt ist und **`y`** anderenfalls. Es handelt sich also um eine Art **`if`**-Anweisung innerhalb von Ausdrücken.

4. Kontrollstrukturen

Mehrfachauswahl

Bei mehrfacher Verzweigung aufgrund eines ganzzahligen Wertes empfiehlt sich die **switch**-Anweisung:

```
switch( Ausdruck ){
    case Konstante_1 :
        Block_1
        break;
    ...
    case Konstante_n :
        Block_n
        break;
    default :  /* optional */
        Block_default
}
```

Die Konstanten müssen ganzzahlig und paarweise verschieden sein. Stimmt eine Konstante mit dem Wert des Ausdrucks überein, so wird der darauf folgende Anweisungsblock ausgeführt.

4. Kontrollstrukturen

Mehrfachauswahl

`break` dient zum Verlassen der **`switch`**-Anweisung. Wird eine der **`break`**-Anweisungen weggelassen und der unmittelbar davor liegende Block wurde abgearbeitet, so wird auch noch der nachfolgende Block ausgeführt.

Die **`switch`**-Anweisung ersetzt folgende Kette von **`if`**-Anweisungen:

```
int i = Ausdruck;
if( i == Konstante_1 ){
    Block_1
} else if( ... ){
    ...
} else if( i == Konstante_n ){
    Block_n
} else {
    Block_default
}
```

4. Kontrollstrukturen

Iterationsabbruch

Die innerste Schleife einer geschachtelten Iteration kann mit **`break`** inmitten des Anweisungsblocks verlassen werden.

Mit **`continue`** kann der Rest eines Anweisungsblocks übersprungen werden. **`break`** und **`continue`** sollten jedoch, soweit möglich, vermieden werden um zu übersichtlichen Implementierungen zu gelangen.

Besteht der Anweisungsblock einer Schleife oder Verzweigung nur aus einer Anweisung, so kann man die Klammern **`{}`** weglassen (es empfiehlt sich jedoch, diese beizubehalten).

4. Kontrollstrukturen

Funktionen

Die **Definition** einer Funktion hat folgende Syntax:

```
RückgabeTyp FunktionsName (
    Typ_1 Arg_1,
    ...
    Typ_n Arg_n
)
{ Block }
```

Die in einem C-Modul definierten Funktionen sollten (mit Ausnahme von main) am Anfang des Moduls deklariert werden. Die **Deklaration** ist identisch mit der Funktionsdefinition, bis auf die Tatsache, dass `{ Block }` durch ein Semikolon ersetzt wird.

4. Kontrollstrukturen

Funktionen

Deklaration:

```
RückgabeTyp FunktionsName (
    Typ_1 Arg_1,
    ...
    Typ_n Arg_n
);
```

Eine Deklaration legt bereits Funktionsname, Rückgabetyp und die Typen der Argumente fest. Die Bezeichner der Argumente *Arg_1*, ..., *Arg_n* haben in der Deklaration keine Bedeutung, es sei denn, es werden Default-Argumente zugewiesen.

Bei Verwendung mehrerer C/C++ Module stehen die Deklarationen in einer gesonderten **Header-Datei** mit der Endung **.h** .

4. Kontrollstrukturen

Globale Variablen

Globale Variablen werden ausserhalb von Funktionen (z.B. direkt nach den **`#include`**-Anweisungen) deklariert. Der Gültigkeitsbereich erstreckt sich dann von der Deklaration bis zum Ende des C-Moduls. Die optionale Wertezuweisung wird nur einmal bei Programmstart ausgeführt.

Hinweis: Globale Variablen sollten aus Gründen der Übersichtlichkeit so wenig wie möglich eingesetzt werden.

4. Kontrollstrukturen

Statische Variablen

Der Wert einer lokalen Variablen ist nach Verlassen der Funktion verloren. Benötigt man diesen Wert bei einem erneuten Funktionsaufruf, so sind statische Variablen einzusetzen. Diese werden mit dem Schlüsselwort **`static`** deklariert. Beispiel:

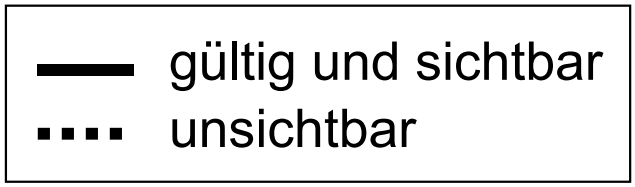
```
void Count(){
     static int n=1;

     cout << "Aufruf Nr. " << n << endl;
     n++;
}
```

Diese Funktion gibt aus, wie oft sie schon aufgerufen wurde. Die Wertzuweisung innerhalb der Deklaration wird bei statischen Variablen nur beim ersten Funktionsaufruf ausgeführt.

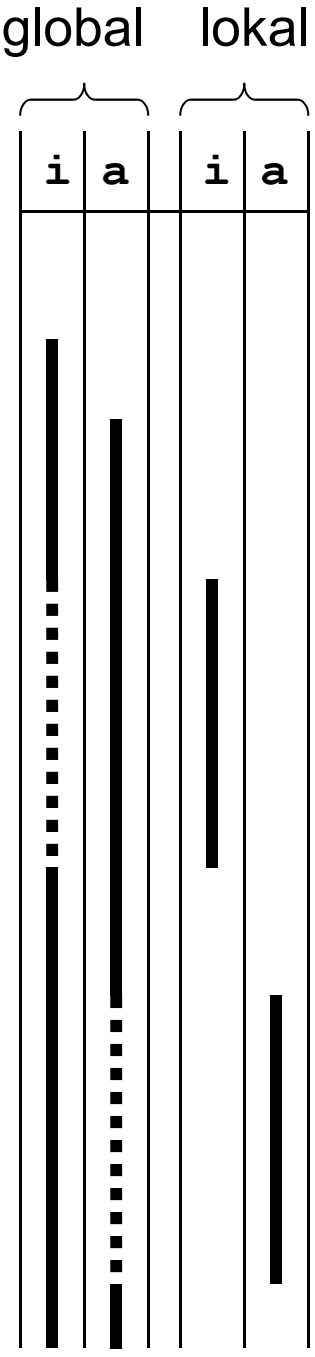
4. Kontrollstrukturen

Gültigkeitsbereiche

Beispiel:




```
#include<iostream>
int i=8;
float a;

void Hello(){
     int i;
     for( i=0; i<3; i++)
          std::cout << "hello\n";
}

void main(){
     static int a=3;
     Hello();
     std::cout << i << "," << a;
     std::cout << std::endl;
}
```



4. Kontrollstrukturen

Konstanten und Makros

Konstanten und Makros können durch die Preprozessor-Anweisung **`#define`** festgelegt werden. Beispiel:

```
#define PI 3.14
#define MAX(A, B) ((A) > (B) ? (A) : (B))
```

Der Unterschied zu einer Variablen- oder Funktionsdefinition besteht darin, dass die mit **`#define`** erzeugten Makros schon beim Compilieren im Programm **textuell ersetzt**, und nicht erst zur Laufzeit ausgeführt werden. Die Programmzeilen

```
Area = PI * Radius * Radius;
MaxFläche = MAX( Area + x, y * z);
```

würden vom Compiler also wie folgt interpretiert:

```
Area = 3.14 * Radius * Radius;
MaxFläche = ((Area + x) > (y * z) ?
     (Area + x) : (y * z));
```

4. Kontrollstrukturen

Konstanten und Makros

Bei der Definition von Makros sollten die Argumente in Klammern gesetzt werden, um Fehler zu vermeiden:

```
#define SQUARE(X) X * X  /* falsch! */
a = SQUARE( z + 1);
```

würde inkorrekt umgesetzt, da die Klammerung fehlt:

```
a = z + 1 * z + 1;
```

Bei Argumenten mit Seiteneffekt (z.B. **`x++`**) ist zu berücksichtigen, dass diese ggf. mehrmals ausgewertet werden.

Achtung: die mit **`#define`** festgelegte Definition ist auf eine Zeile beschränkt und kann aus mehreren Zeichenfolgen bestehen (es folgt kein Semikolon).

Für den Namen eines Makros sollten Grossbuchstaben verwendet werden, um diesen besser von einem Funktionsnamen unterscheiden zu können.

4. Kontrollstrukturen

Konstanten und Makros

Definitionen können auch erfolgen, um bestimmte Programmblöcke ein- oder auszublenden. Beispiel:

```
#define ECHO
...
#ifdef ECHO
    printf("Ergebnis: %5d\n", Ergebnis);
#endif
```

Durch Auskommentieren der ersten Zeile (also `// #define ECHO`) kann man erreichen, dass bestimmte Blöcke im nachfolgenden Programmtext vom Compiler ignoriert werden.

Auch in Header-Dateien ist ein derartiges Konstrukt sinnvoll, um mehrfaches Compilieren des Inhalts zu vermeiden:

```
#ifndef MY_HEADER  /* falls nicht def. */
   #define MY_HEADER
   ...  /* Inhalt der Header-Datei */
#endif
```

5. Zeiger und Arrays

Zeiger

Variablen werden in C und C++ nicht nur eingesetzt um Daten zu speichern, sondern auch **Adressen**.

Die Adresse einer Variablen gibt an, wo der zugewiesene Wert im Speicher abgelegt ist. Diese Adresse kann mit dem unären Operator **`&`** abgefragt werden und ist vom Typ **`(`***Variablentyp*`*)`**. Beispiel:

```
int i = 5;
int *Zeiger = &i; /* zeigt auf die 5 */
```

In diesem Beispiel enthält die Variable **`Zeiger`** vom Typ **`(int*)`** die Adresse des abgelegten Datums **`5`** der Variablen **`i`**.

5. Zeiger und Arrays

Zeiger

Die Deklaration von Variablen und Zeigern auf den selben Typ kann auch kombiniert werden:

```
int i = 5, *Zeiger = &i, j, **k;
```

Der jeweilige Typ setzt sich dabei aus dem Basistyp (hier: **int**) und den möglichen ***-Zusätzen direkt vor der Variablen zusammen. Die obige Deklaration ist äquivalent mit

```
int i = 5;
int *Zeiger = &i;
int j;
int **k;
```

Die Variable **k** ist hier ein Zeiger auf den Typ **(int*)**. Eine legitime Zuweisung wäre also

```
k = &Zeiger;
```

Zeiger die (noch) nicht gebraucht werden können mit dem Wert **NULL** initialisiert werden

5. Zeiger und Arrays

Zeiger

Die Umkehrung des Adress- (oder Referenz-) Operators **&** ist durch den unären Dereferenz-Operator ***** realisiert. Beispiel:

```
float f, *Zeiger = &f;
f = 2.0;
cout << *Zeiger << endl; // 2.0
```

Hier hat die Variable **f** zwischenzeitlich den Wert **2.0** angenommen. Da **Zeiger** die Adresse dieses Wertes enthält, die während des gesamten Gültigkeitsbereiches von **f** erhalten bleibt, liefert ***Zeiger** den aktuellen Wert von **f**.

Hinweis: Die unären Operatoren **&** und ***** können auch in arithmetischen und logischen Ausdrücken verwendet werden und haben dabei größere Priorität als arithmetische und logische Operatoren (außer **--** und **++** und).

5. Zeiger und Arrays

Zeiger

Ist der Basistyp eines Zeigers unbekannt, so kann man den Typ **(void*)** verwenden und bei einer Zuweisung eine Typumwandlung vornehmen. Alle Zeiger benötigen gleich viel Speicherplatz (4 Bytes). Im Gegensatz zu arithmetischen Ausdrücken ist für Zeiger keine automatische Typumwandlung realisiert.

Beispiel:

```
int   i, *pi = &i, j, k;
float f = 3.0, g;
void *p;

*pi = 5;                // i = 5
j = *pi * *pi;          // j = 25
p = (void*)&j;          // p-->j
k = *(int*)p;           // k = 25
p = (void*)&f;          // p-->f
g = i + *(float*)p;     // g = 8.0
```

5. Zeiger und Arrays

Call by Value

Bei Funktionsaufrufen in C wird der Wert der Argumente an lokale Variablen weitergegeben, deren Gültigkeitsbereich auf die aufgerufene Funktion beschränkt ist (Call by Value).

Beispiel:

```
void Mystery( int a, int b){
    int h;
    h = a; a = b; b = h;
}

int main(){
    int i = 2, j = 7;

    Mystery( i, j);
    cout << i << "," << j << endl;
}
```

Der Aufruf von **`Mystery`** hat keinen Effekt, da nur die Werte lokaler Variablen verändert werden.

5. Zeiger und Arrays

Call by Reference

Übergibt man an Stelle der Werte die Adressen der Argument-Variablen (wie z.B. bei **`scanf`**), so ist eine Manipulation der Werte innerhalb der Funktion möglich. (Call by Reference). Beispiel:

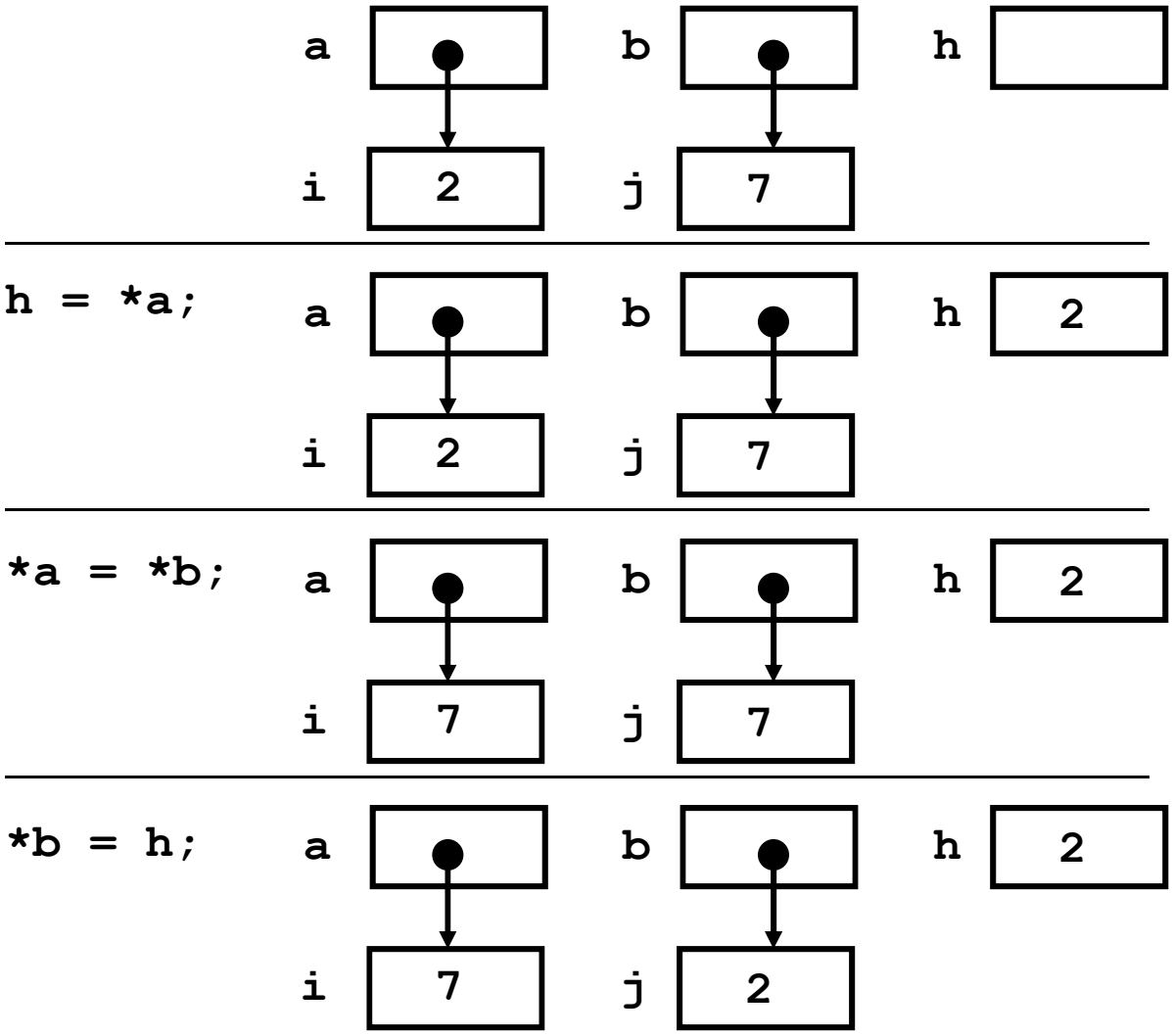
```
void Swap( int *a, int *b){
    int h;
    h = *a; *a = *b; *b = h;
}

int main(){
    int i = 2, j = 7;

    Swap( &i, &j);
    cout << i << "," << j << endl;
}
```

Da die Werte von **`i`** und **`j`** vertauscht wurden, lautet die Ausgabe: **`7, 2`**

5. Zeiger und Arrays

Call by Reference

Es folgt eine Übersicht der Variablen für den Aufruf **`Swap( &i, &j);`**

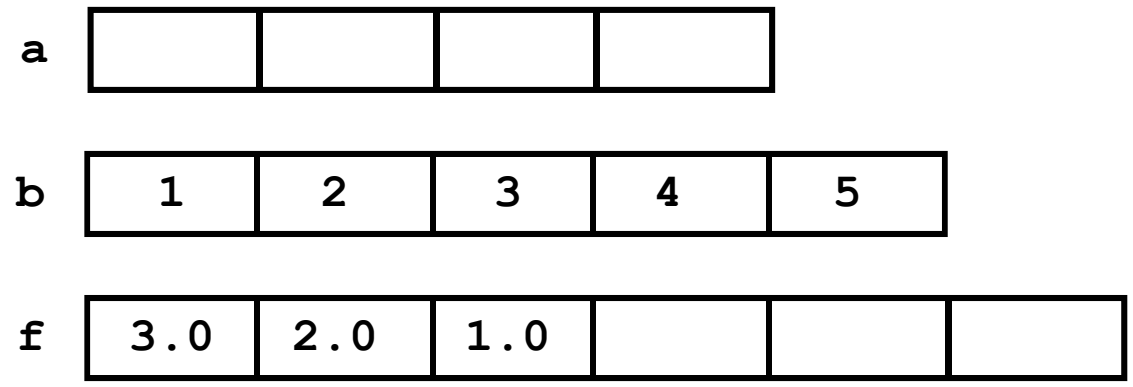


5. Zeiger und Arrays

Arrays

Die Größe eines Arrays (Variablenfeldes) wird bei der Deklaration festgelegt (muss konstant sein). Die Zuweisung einer (Teil-)Liste von Werten ist optional. Beispiel:

```
int    a[4], b[] = {1,2,3,4,5};
double f[6] = {3.0, 2.0, 1.0};
```



Die Größe von **b** wird hier durch die Werteliste bestimmt. Die einzelnen Elemente eines Arrays können mit dem **[]**-Operator erreicht werden, z.B. **b[0] =1**, **b[1] =2** ,... , **b[4] =5**.

5. Zeiger und Arrays

Arrays

Wird ein Array der Größe ***n*** deklariert, so laufen die Indizes von **0** bis ***n*-1**. Der Compiler überprüft nicht, ob diese Grenzen eingehalten werden. Versucht man z.B. einem Element mit Index größer als ***n*-1** einen Wert zuzuweisen, so kann es entweder zum Programmabsturz kommen, oder es können andere Daten, die im Speicher in der Nähe des Arrays abgelegt sind, beschädigt werden.

Beispiel:

```
int    i, a[10];

for( i=1; i<=10; i++){
       a[i] = i;
       // Fehler: a[10] existiert nicht!
}
```

5. Zeiger und Arrays

Arrays

Arrays werden intern wie Zeiger auf den Elementtyp behandelt. Der Wert eines Arrays ist daher die Adresse des ersten Elements. Im Gegensatz zu Zeigern darf dieser Wert jedoch nicht verändert werden.

Beispiel:

```
double d, a[3] = {3.0, 8.0, 1.0};
double *p;
p = a;              // p-->a[0]
d = *p;             // d = 3.0
p = &a[1];          // p-->a[1]
d = *p;             // d = 8.0
p++;                // p-->a[2]
d = *p;             // d = 1.0
a = p;              // Fehler !!!
```

Hinweis: die Operatoren `--` und `++` verändern den Wert eines Zeigers nicht um eins, sondern um die Größe des Basistyps.

5. Zeiger und Arrays

Arrays

Der Gültigkeitsbereich von Arrays kann lokal oder global definiert sein, analog zu einfachen Variablen. Es ist zu berücksichtigen, dass lokale Arrays bei jedem Funktionsaufruf neu angelegt werden (sofern sie nicht statisch sind) und nur während der Abarbeitung dieses Aufrufs gültig sind. Beispiel:

```
int *GetArray(){  // falsch!!!
    int a[5] = {1,2,3,4,5};
    return a;
}

int main(){
    int *p;
    p = GetArray();
    cout << p[3] << endl; // Fehler!
    // die Adresse p ist ungültig
}
```

Achtung: Fehler dieser Art werden nicht vom Compiler erkannt. Manchmal tritt nicht mal ein Fehler auf, wenn der ungültige Speicherbereich unbelegt ist.

5. Zeiger und Arrays

Dynamische Arrays

Es ist auch möglich, während der Laufzeit eines C-Programms Speicherbereiche dynamisch zu reservieren und danach wieder freizugeben, z.B. wenn die Größe eines Arrays erst zur Laufzeit feststeht.

In C ist das Allokieren sehr umständlich: Die Funktion

```
void *malloc( size_t size)
```

in der Bibliothek **`stdlib.h`** reserviert einen Speicherbereich definierbarer Größe (in Bytes) und gibt die Anfangsadresse zurück (oder **`NULL`**, falls nicht genug Speicher verfügbar ist). Mit der Funktion

```
void free( void *p)
```

kann der reservierte Speicherbereich dann wieder freigegeben werden.

In C++ sollte man man **`new`** und **`delete`** verwenden, z.B.

```
int *a = new int; ... delete a;
int *b = new int[n]; ... delete[] b;
```

5. Zeiger und Arrays

Dynamische Arrays

Beispiel:

```
int *a = new int[n];
a[0] = ...
...
a[n-1] = ...
...
delete[] a;
```

liefert ein Array der Länge ***n***, analog zu

```
int a[n];
```

Der Unterschied besteht darin, dass der Gültigkeitsbereich erst dann endet, wenn

```
delete[] a;
```

aufgerufen wird. Diese Variante des Anlegens von Arrays zur Laufzeit heißt **dynamisch** (im Gegensatz zu **statischen** Arrays, deren Größe und Gültigkeitsbereich im Programm fest verankert ist).

5. Zeiger und Arrays

Dynamische Arrays

Mit Hilfe von dynamischen Speicherbereichen lassen sich Kreatoren für das Anlegen und Initialisieren von Arrays implementieren. Beispiel:

```
float *GetFloatArray( int n){
  /* Liefert ein dynamisches Array
     vom Typ float und initialisiert
     die Elemente mit 0.0 */

    float *a = new float[n];

    if( a != NULL){
        for( i=0; i<n; i++) a[i] = 0.0;
    }
    return a;
}
```

5. Zeiger und Arrays

Vector

In C++ empfiehlt sich die Verwendung des Vector-Templates für dynamische Arrays, z.B.:

```
#include<vector>
...
vector<int> f(5); // 5 integers
f[3] = 4;
f.at(4) = 5;       // f[4] = 5
f.resize(7);
f[6] = 10;
```

Der Operator `[]` überprüft im Gegensatz zu `at()` nicht den Bereich und ist daher anfälliger für Fehler (wenngleich effizienter).

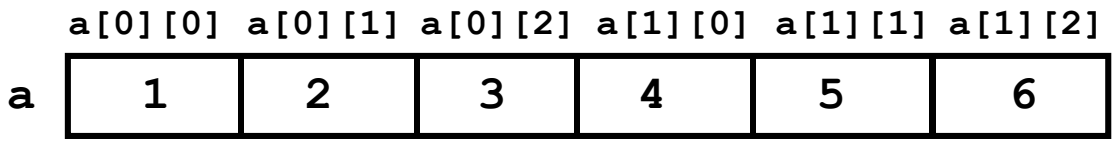
5. Zeiger und Arrays

Mehrdimensionale Arrays

Mehrdimensionale Arrays werden wie folgt deklariert:

```
int   a[2][3] = {{1,2,3}, {4,5,6}};
float f[3][7][5];
```

die Elemente mehrdimensionaler Arrays werden hintereinander in einem eindimensionalen Array gespeichert:



Setzt man in diesem Beispiel einen Zeiger **`p`** vom Typ **`(int*)`** auf das erste Element von **`a`**, so kann **`a[i][j]`** auch über **`p[i*3 + j]`** erreicht werden.

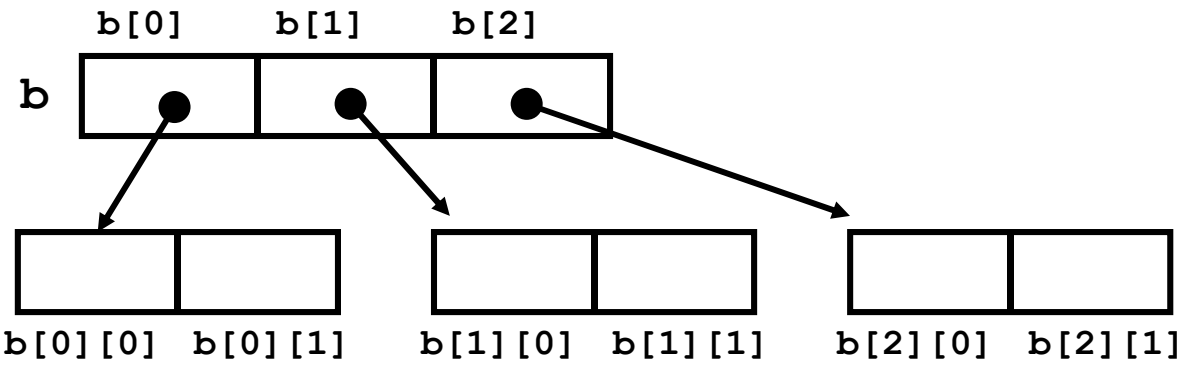
Die erste Dimension ist für die Indizierung nicht relevant und kann, z.B. bei Übergabe des Arrays an eine Funktion, weggelassen werden (**`int a[][3]`** oder **`int (*a)[3]`**).

5. Zeiger und Arrays

Mehrdimensionale Arrays

Alternativ zum statischen Anlegen mehrdimensionaler Arrays kann auch ein Array von Zeigern deklariert werden. Für die einzelnen Elemente ist damit jedoch noch kein Speicherplatz reserviert. Beispiel:

```
int *b[3], i;
for( i=0; i<3; i++){
    b[i] = new int[2];
}
```



Vorteil: die einzelnen Unter-Arrays können verschieden lang sein. Diese Konstruktion ist zwar möglich, aber unüblich.

5. Zeiger und Arrays

Memory Leaks

Wichtig bei der Verwendung von **`new`** ist, dass alle dynamisch erzeugten Objekte irgendwann wieder mit **`delete`** bzw. **`delete[]`** gelöscht werden müssen. Vergisst man dies, so entstehen "memory leaks", d.h. es wird immer mehr Speicher reserviert, bis der Arbeitsspeicher und schlussendlich der Speicher auf der Festplatte nicht mehr ausreicht.

Man sollte daher nur sparsam von der Möglichkeit des dynamischen Allokierens gebrauch machen. In C++ empfiehlt sich die Anwendung fertiger Datenstrukturen in Form der Templates

`array`, **`vector`**, **`list`**, **`queue`**, ...

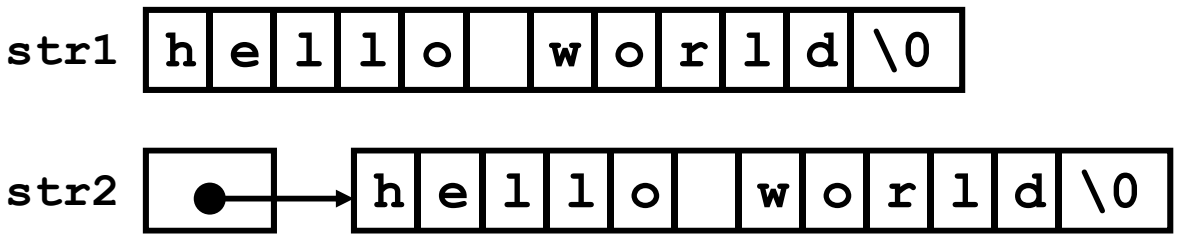
um dynamisches Allokieren mit **`new`** und **`delete`** gänzlich zu umgehen.

5. Zeiger und Arrays

Zeichenketten

Zeichenketten (Strings), die in C z.B. mit **`printf`** ausgegeben werden, sind Arrays vom Basistyp **`char`**. Beispiele für eine Deklaration mit Zuweisung:

```
char str1[] = "hello world";
char *str2 = "hello world";
```



Zeichenketten enden mit `'\0'`, auch wenn das verwendete Array noch mehr Zeichen unterbringen kann. Beim Reservieren von Speicher für eine Zeichenkette muss also der Platz für ein Zeichen addiert werden.

5. Zeiger und Arrays

Zeichenketten

Zeichenketten können in C mit dem Steuerzeichen **"%s"** mit **scanf** eingelesen werden. Im Gegensatz zum Einlesen von Basistypen entfällt der Adressoperator **&**, da das übergebene Argument bereits von einem Zeigertyp ist:

```
char str[1024]; /* Platz für 1023
                   Zeichen */
scanf("%s", str);  /* Eingabe ohne & */
printf("%s", str); /* Ausgabe */
```

Hierbei ist zu berücksichtigen, dass der Benutzer nicht mehr als 1023 Zeichen eingeben darf, da es sonst zu einer Überschreitung des reservierten Bereichs kommt (das letzte Zeichen ist **\0**).

Hinweis: **'c'** ist ein **char**, **"c"** jedoch ein **char[]** .

5. Zeiger und Arrays

Zeichenketten

In C++ verwendet man die Bibliothek **`string`** mit dem gleichnamigen Datentyp im Namespace **`std`**. Beispiel:

```
std::string s = "Hello";
s += " World!\n";
std::cout << s;
// Ausgabe: Hello World!
int i = s.find( "World");
char c = s[i];   // oder s.at(i);
std::cout << "c = \'" << c << "\'\n";
// Ausgabe: c = 'W'
s = "Aachen" < "Aurich" ? "yes" : "no";
std::cout << s << "\n"; // yes
```

Man sieht, dass sich die Klasse **`string`** genau wie ein Array von Typ **`char`** verhält, aber noch zusätzliche Funktionen, wie etwa das Suchen und Vergleichen enthält.

5. Zeiger und Arrays

Zeichenketten

Arrays von Zeichenketten können in C wie folgt definiert werden (analog in C++ mit **`string`**):

```
char *Monate[12] = {
    "Januar", "Februar", "Maerz",
    ... , "Dezember"
    };
```

Hierbei handelt es sich um ein Array von 12 Zeigern auf die einzelnen Zeichenketten.

Analog kann man auch ein zweidimensionales Array vom Typ **`char`** definieren:

```
char Monate[][20] = {
    "Januar", "Februar", "Maerz",
    ... , "Dezember"
    };
```

In diesem Fall wird jede Zeichenkette in einem Unter-Array der Länge 20 abgelegt.

5. Zeiger und Arrays

Argumente von main

Beim Aufrufen eines compilierten C-Programms von der Konsole kann der Benutzer Argumente übergeben. Diese werden in einem Array von Zeigern auf Zeichenketten an die Funktion **`main`** übergeben. Die erste Zeichenkette enthält dabei den Namen des aufgerufenen Programms, z.B. **`"a.out"`**. Die einzelnen Argumente kann man wie folgt ausgegeben:

```
#include<stdio.h>
int main( int argc, char *argv[]){
    /* argc : Anzahl der Argumente */
    /* argv : Array mit Argumenten */
    int i;

    for( i=0; i<argc; i++){
        printf("Arg. %5d : %s\n",
            i, argv[i]);
    }
}
```

5. Zeiger und Arrays

Sortieren

Das Sortieren von Zahlen und Zeichenketten ist ein häufig auftretendes Problem. Es werden im Folgenden C-Implementierungen für Bubblesort und Mergesort vorgestellt.

Eingabe: Array von Zahlen **`a[i] (i=0,...,n-1)`**.

Ausgabe: **`b`** ist eine Permutation von **`a`** mit **`b[i]`** $\leq$ **`b[i+1]`** für **`i=0,...,n-2`**.

Das Vertauschen von Elementen kann über eine Funktion **`Swap( int *i, int *j)`** realisiert werden (Call-by-Reference). Bei der Anpassung an andere Datentypen ist im Wesentlichen nur diese Funktion zu ersetzen.

5. Zeiger und Arrays

Bubblesort

Jeder Durchgang permutiert ein Unterarray **a[0]...a[i-1]**, so dass das größte Element den Index **i-1** besitzt. Dies kann wie folgt implementiert werden:

```
for( j=0; j<i-1; j++){   /* j=0...i-2 */
    if( a[j] > a[j+1]){
        /* vertausche a[j] mit a[j+1] */
        Swap( &a[j], &a[j+1]);
    }
}
```

Dieser Programmteil wird zuerst mit **i=n** durchgeführt, dann mit **i=n-1**, usw., bis **i=2**. Es sind also zwei geschachtelte Schleifen zu durchlaufen:

5. Zeiger und Arrays

Bubblesort

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

// Funktionsdeklarationen
void Swap( int *i, int *j);
void BubbleSort( int n, int a[]);

// Vertauschen zweier Zahlen
void Swap( int *i, int *j){
    int tmp = *i;
    *i = *j;
    *j = tmp;
}
```

5. Zeiger und Arrays

Bubblesort

```
// Sortieren eines int-Arrays
void BubbleSort( int n, int a[]){
    int i, j;

    // i = n-2...2
    for( i=n-2; i>1; i--){

        // j = 0...i-2
        for( j=0; j<i-1; j++){

            // ggf. vertauschen
            if( a[j] > a[j+1]){
                Swap( &a[j], &a[j+1]);
            }
        }
    }
}
```

5. Zeiger und Arrays

Mergesort

Mergesort bietet einen effizienteren Ansatz mit $O(n \log_2 n)$ Operationen (Divide-and-Conquer). Das Array wird in zwei gleich große Teile zerlegt, die rekursiv durch das selbe Verfahren sortiert werden (Rekursionsabbruch bei n=1). Anschließend werden die sortierten Teilarrays im Reißverschluss-Verfahren zusammengefügt:

```
void MergeSort( int n, int a[]){
    int i, j, k = n/2;
    int *b = new int[n];

    // Rekursionsabbruch
    if( n <= 1) return;
```

5. Zeiger und Arrays

Mergesort

```
    // Sortiere die zwei Teillisten
    MergeSort( k, a);
    MergeSort( n-k, &a[k]);

    // kopiere Teillisten a nach b
    for( i=0; i<n; i++) b[i] = a[i];

    // Zusammenfuegen der Teillisten
    j=0;
    for( i=0; i<n; i++){
        if( j >= n/2)       a[i] = b[k++];
        else if( k >= n)    a[i] = b[j++];
        else if(b[j]<b[k]) a[i] = b[j++];
        else                a[i] = b[k++];
    }
    delete[] b; // Freigeben von b
}
```

6. Datenstrukturen

Typdefinitionen

Neben den in C verfügbaren Basistypen können mit **`typedef`** weitere Variablentypen definiert werden, z.B.

```
typedef int Laenge;
typedef char *String;
typedef double Vektor[3];
typedef double Matrix[3][3];
```

Es ist zu beachten, dass der mit **`typedef`** vereinbarte Typbezeichner in der Position des Variablennamens (und nicht unmittelbar nach **`typedef`**) auftritt. Die vereinbarten Datentypen können dann im nachfolgenden Programm eingesetzt werden, z.B.

```
Laenge  Kathete = 3, Hypotenuse = 5;
String  c = "Hello";
Vektor  x = { 1.0, 0.0, 0.0};
Matrix  A, B;
```

6. Datenstrukturen

Aufzählungstypen

Aufzählungstypen werden eingesetzt, um bestimmte Zustände (Zahlenwerte) zu benennen:

typedef enum {*b0,b1,...,bn***}** *typname;*

Wobei *b0,b1,...,bn* und *typname* Bezeichner sind. Beispiel:

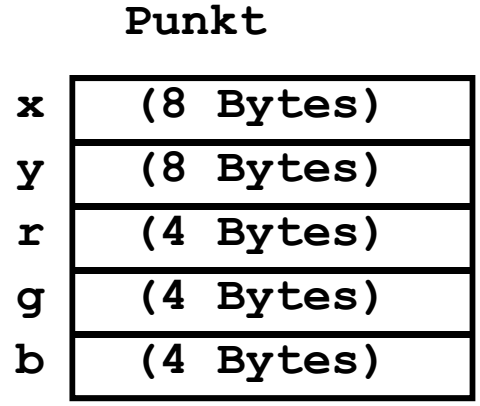
```
typedef enum {Januar, Februar, ...,
        Dezember} Monat;

Monat m = Januar;
```

(**m** wird intern als Zahl dargestellt.)

6. Datenstrukturen

Strukturen und Klassen

```
class Punkt{        // C++
    double x, y;   // Koordinaten
    int r, g, b;   // Farbanteile
};

typedef struct {   /* wie oben in C */
    double x, y;   /* Koordinaten */
    int r, g, b;   /* Farbanteile */
} Punkt;
```

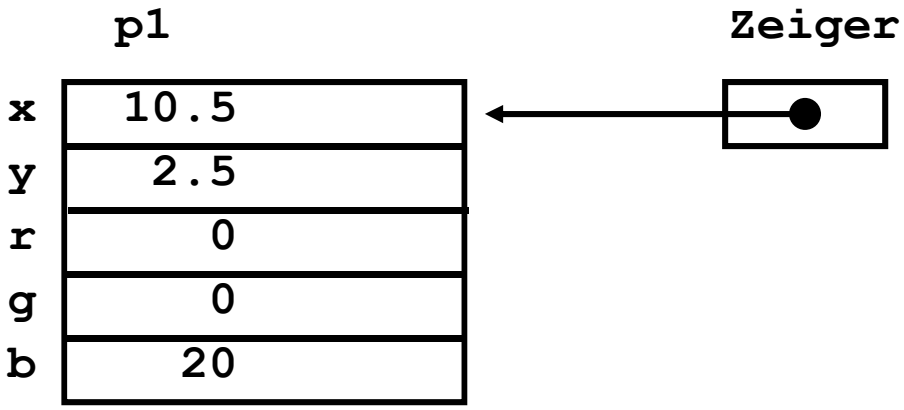


6. Datenstrukturen

Strukturen

Die einzelnen Datenelemente einer Struktur / Klasse können mit den Operatoren `.` und `->` erreicht werden, je nach dem, ob eine **Instanz** oder ein **Zeiger** auf die Klasse vorliegt. Beispiel:

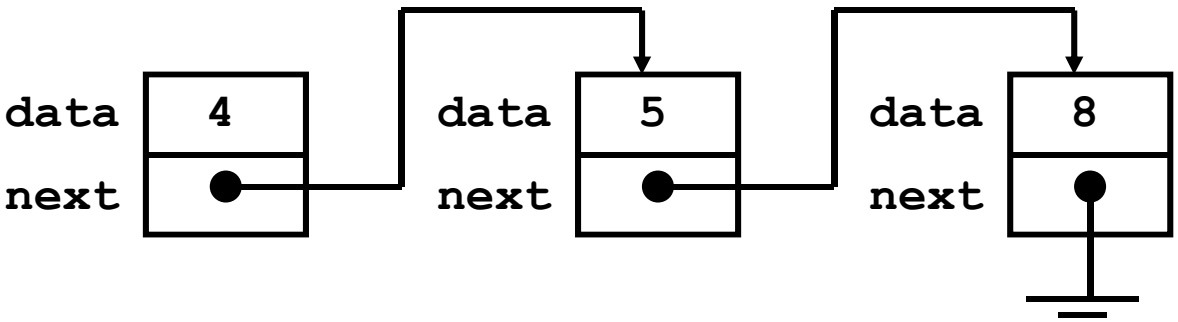
```
Punkt  p1, *Zeiger = &p1;
p1.x = 10.5;
Zeiger->y = 2.5;
p1.r = p1.g = 0;
Zeiger->b = 20;
```



6. Datenstrukturen

Verkettete Listen in C

Eine Klasse kann Zeiger auf eigene Instanzen enthalten. Beispiel: Eine einfach verkettete Liste




```
class Liste{      // C++
    int data;     // Datenelement
    Liste *next; // Zeiger
};

/* in C etwas umstaendlicher */
typedef struct Element{
    int data;         /* Datenelement */
    struct Element *next; /* Zeiger */
} Liste;
```

6. Datenstrukturen

Verkettete Listen in C

Auch in C kann man Konstruktoren programmieren:

```
Liste *NeuesElement( int data){
    /* Neues Listenelement Anlegen */
    Liste *elem = (Liste*)
        malloc( sizeof( Liste));

    elem->data = data;
    elem->next = NULL;
    return elem;
}
```

Will man nun ein Element am Listenanfang einfügen, ändert sich ja der Zeiger auf das erste Element. Da man den geänderten Zeiger auch noch zurückgeben muss, braucht man einen Zeiger auf einen Zeiger, was in C zu besonderen "Stilblüten" führt:

6. Datenstrukturen

Verkettete Listen in C

```
void Einfuegen( Liste **ppl, int data){
    /* Einfuegen am Listenanfang */
    Liste *elem = NeuesElement( data);

    /* Liste an elem anhaengen */
    elem->next = *ppl;
    *ppl = elem; /* Listenkopf */
    return;
}

void PrintListe( Liste *pl){
    /* Ausgeben der Liste */
    if( pl==NULL) return; /* leer */

    printf("%5d\n", pl->data);
    PrintListe( pl->next); /* Rekursion*/
}
```

6. Datenstrukturen

Verkettete Listen in C

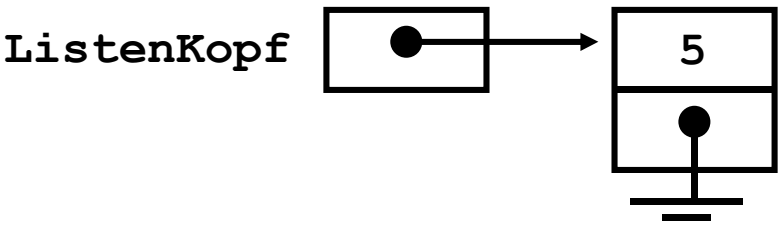
```
int main(){
    Liste *ListenKopf = NULL;
    int    data;

    do{
        printf(" Eingabe (0=Ende):\n");
        scanf("%d", &data);

        Einfuegen( &ListenKopf, data);
        PrintListe( ListenKopf);

    /* Abbruch bei Eingabe von 0 */
    } while (data == 0);
}
```

6. Datenstrukturen

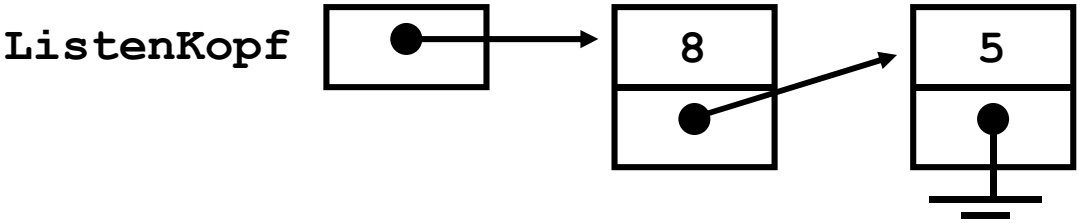
Verkettete Listen in C

Testlauf:


```
Einfuegen( &Listenkopf, 5);
```




```
Einfuegen( &Listenkopf, 8);
```



6. Datenstrukturen

Listen in C++

In C++ muss man sich mit solchen Datenstrukturen nicht abmühen, da diese bereits als Template vorhanden sind und somit für jeden beliebigen Basistyp einsetzbar sind.

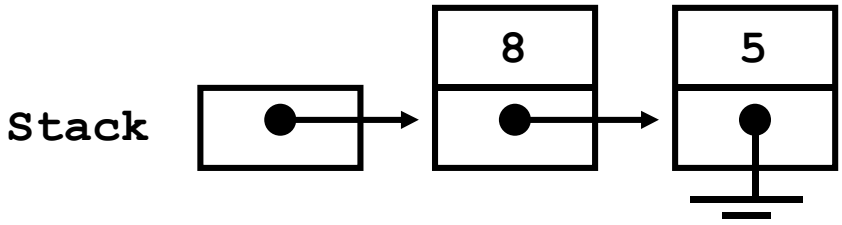
```
#include<list>
...
int data[] = {3,5,2,4};
// Uebernahme der Daten und sortieren
list<int> Liste( data, data+4);
Liste.sort();

for( list<int>::iterator i =
     Liste.begin(); i!=Liste.end(); i++){
    std::cout << *i << "\n"; // Ausgabe
}
```

Hinweis: data+4 ist ein Zeiger, auf das fünfte Element (wenn es eins gäbe). Mit dem **Iterator** kann man die Liste bequem abarbeiten.

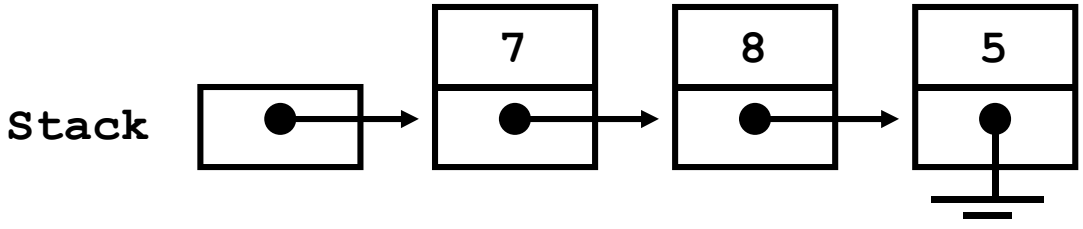
6. Datenstrukturen

Stack (FIFO)

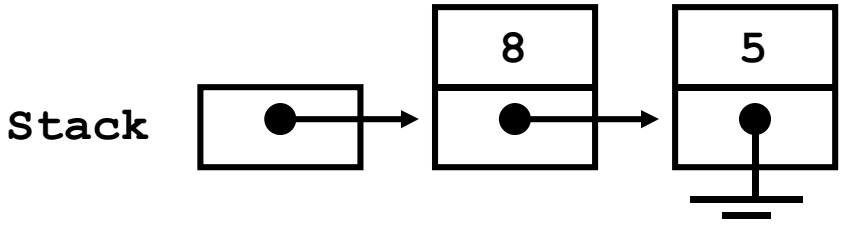
Häufig benötigt man eine **FIFO-Liste** (first in first out) oder einen **Stack** (Stapel) als Datenstruktur. Die Elemente werden mit der Operation **`push`** auf den Stack gelegt und mit **`pop`** in umgekehrter Reihenfolge wieder heruntergenommen. Beispiel:




```
Stack.push( 7);
```




```
... = Stack.top(); Stack.pop()
```



6. Datenstrukturen

Stack (FIFO)

Als Datentyp wird das Template **stack** verwendet:

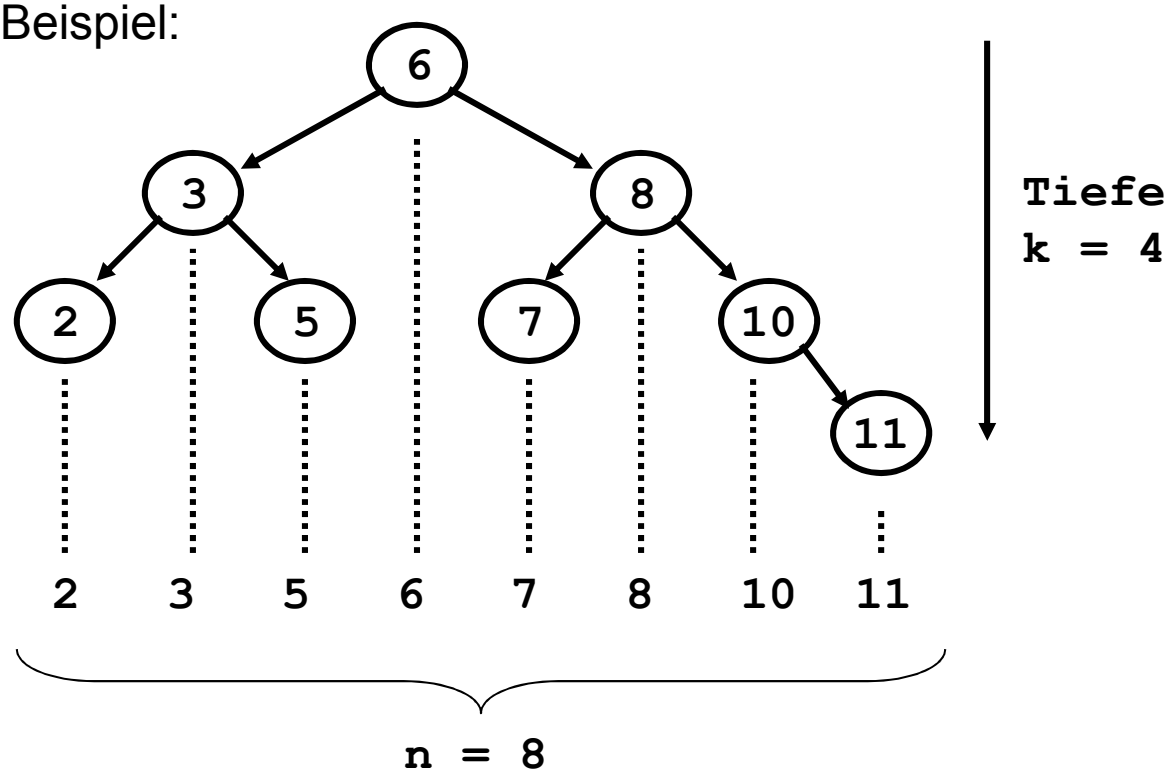
```
#include<stack>
...
int data[] = {3,5,2,4};
stack<int> Stack;

// Daten auf Stack ablegen
for (int i=0; i<4; i++){
    Stack.push( data[i]);
}

// ... und wieder herunternehmen
while ( !Stack.empty()){
    std::cout << Stack.top() << "\n";
    Stack.pop();
}
```

6. Datenstrukturen

Binärbaum

Eine sortierte Liste kann ebenfalls in einem Binärbaum (in-order) untergebracht sein. Der Vorteil gegenüber einer verketteten Liste besteht darin, dass einzelne Elemente schneller einsortiert bzw. gefunden werden können, sofern der Baum relativ ausgewogen ist.

Beispiel:

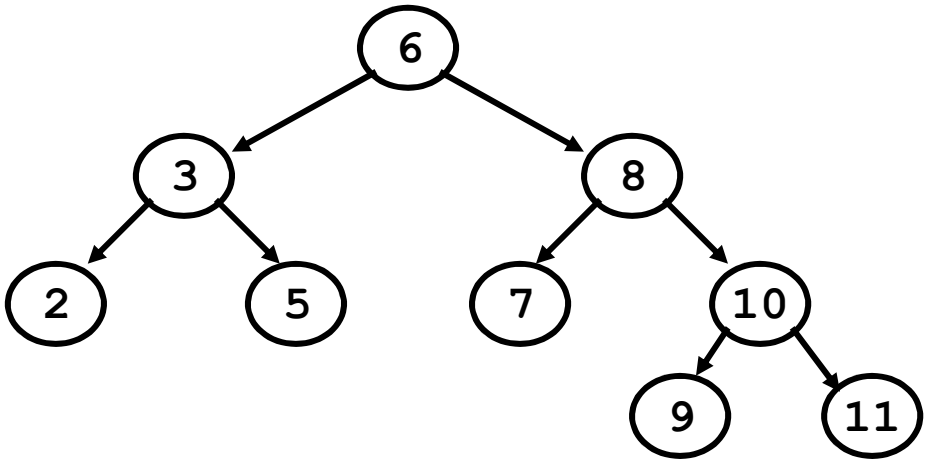


6. Datenstrukturen

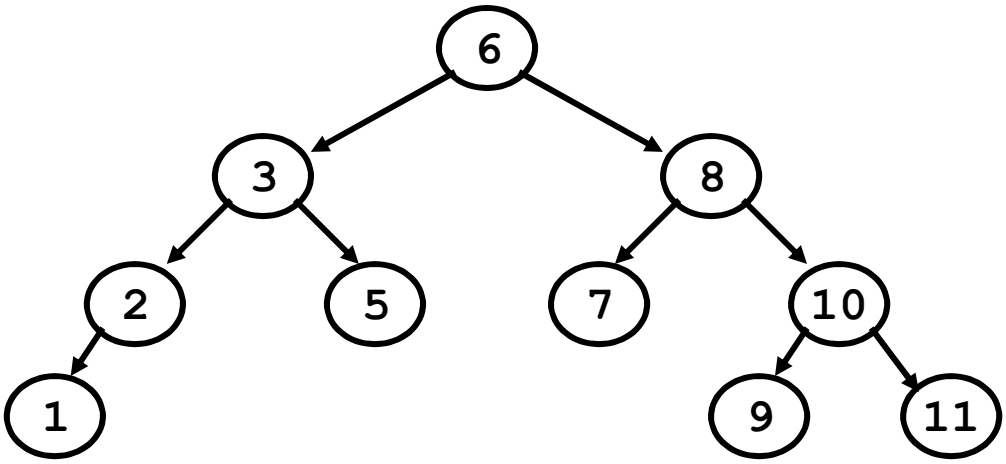
Binärbaum

Beispiel (Einsortieren):

```
-> 9
```




```
-> 1
```



6. Datenstrukturen

Binärbaum

Es folgt eine Implementierung für sortierte Binärbäume.

```
class TreeNode; // Deklaration

// Hauptklasse fuer Baum
class Tree{
private:
    int n_nodes;    // Anzahl der Knoten
    TreeNode *root; // Wurzel

public:
    Tree(); // Konstruktor
    ~Tree(); // Destruktor

    void Insert( int data); // Einfuegen
    void Print(); // sortierte Ausgabe
};
```

6. Datenstrukturen

Binärbaum

```
// Hilfsklasse fuer Baumknoten
class TreeNode{
private:
    int data;
    TreeNode *left, *right;

public:
    TreeNode(); // Konstruktoren
    TreeNode( int data);

    ~TreeNode(); // Destruktor

    void Insert( int data); // Einfuegen
    void Print(); // Ausgeben
};
```

6. Datenstrukturen

Binärbaum

```
Tree::Tree(){ // Konstruktor
       n_nodes = 0; // Anzahl Knoten = 0
       root = NULL; // Wurzel erden!
}

Tree::~Tree(){ // Destruktor
    // rekursives Loeschen aller Knoten
    if (root != NULL) delete root;
}

void Tree::Insert( int data){// Einfuegen
    n_nodes++;
    // Wurzel anlegen bzw.
    // rekursives Einfuegen
    if( root == NULL){
        root = new TreeNode( data);
    }else{
        root->Insert( data);
    }
}
```

6. Datenstrukturen

Binärbaum

```
void Tree::Print(){ // Ausgabe
     if( root == NULL){
          cout << endl << "leerer Baum\n";
     }else{ // rekursive Ausgabe
          root->Print();
     }
     cout << endl;
}

// Baumknoten "in-order" ausgeben
void TreeNode::Print(){
     if( left  != NULL) left->Print();
     cout << data << endl;
     if( right != NULL) right->Print();
}
```

6. Datenstrukturen

Binärbaum

```
TreeNode::TreeNode(){ // Konstruktor
        left = right = NULL; // erden
}

TreeNode::TreeNode( int data){
// Konstruktor mit Datum
        this->data = data;
        left = right = NULL;
}

TreeNode::~TreeNode(){ // Destruktor
        if( left  != NULL) delete left;
        if( right != NULL) delete right;
}
```

6. Datenstrukturen

Binärbaum

```
void TreeNode::Insert( int data){
// Datenelement einsortieren

    if( this->data > data ){
        // in linken Teilbaum einfuegen
        if( left == NULL)
            left = new TreeNode( data);
            else left->Insert( data);
    }else{
        // in rechten Teilbaum einfuegen
        if( right == NULL)
            right = new TreeNode( data);
            else right->Insert( data);
    }
}
```

7. Bibliotheken

C-Umgebung

Neben dem Befehlsumfang der Programmiersprache C legt der **ANSI-Standard** eine Reihe von Datentypen, Funktionen, Makros und Konstanten fest. Diese sind in eigenständigen C-Modulen (**Bibliotheken**) implementiert, welche mit **`#include<`***`header`***`>`** einem C-Programm beigefügt werden können.

Die wichtigsten Definitionen der Standard-Bibliotheken sind in diesem Kapitel zusammengefasst. Es existieren (unter anderem) folgende Header-Dateien:

`stdio.h`	Standard Ein-/Ausgabe
`ctype.h`	Tests für Zeichenklassen
`string.h`	Zeichenketten
`math.h`	Mathematische Funktionen
`stdlib.h`	Hilfsfunktionen
`time.h`	Zeitabfrage
`limits.h`, **`float.h`**	Grenzwerte für Zahlendarstellungen

7. Bibliotheken

Standard Ein-/Ausgabe in C mit `<stdio.h>`

Dateioperationen

```
FILE *stdin, *stdout, *stderr;
```

Ein-/Ausgabe und Fehlermeldungen werden von C wie Dateien (**Datenströme**) behandelt.

```
FILE *fopen( char *filename, char* mode);
```

Öffnen einer Textdatei. **mode** : "**r**" = lesen, "**w**" = überschreiben, "**a**" = anfügen.

```
int fflush( FILE *stream);
```

Gepufferte Daten werden unmittelbar geschrieben.

```
int fclose( FILE *stream);
```

Datei schließen.

```
int remove( char *filename);
int rename( char *filename);
```

Datei löschen bzw. umbenennen.

7. Bibliotheken

Standard Ein-/Ausgabe in C mit `<stdio.h>`

Ausgabe

```
int fprintf( File *stream, char *format,
    ...);
```
Formatierte Ausgabe in eine Datei.

```
int printf( char *format, ...);
```
Äquivalent zu `fprintf( stdout, ...)`.

```
int sprintf( char *s, char *format, ...);
```
Formatierte Ausgabe in eine Zeichenkette `s`.

```
int fputc( int c, FILE *stream);
```
Ausgabe eines Zeichens (Bytes) `c`. Liefert im Fall eines Schreibfehlers den Wert `EOF`.

```
int fputs( char *s, FILE *stream);
```
Ausgabe einer Zeichenkette und "`\n`".

7. Bibliotheken

Standard Ein-/Ausgabe in C mit `<stdio.h>`

Eingabe

```
int fscanf( File *stream, char *format,
    ...);
```

Einlesen aus einer Datei. **Wichtig:** Call-by-Reference.

```
int scanf( char *format, ...);
```

Äquivalent zu `fscanf( stdin, ...)`.

```
int sscanf( char *s, char *format, ...);
```

Einlesen aus einer Zeichenkette `s`.

```
int fgetc( FILE *stream);
```

Einlesen eines Zeichens (Bytes), umgewandelt nach `int`. Liefert bei Dateiende den Wert `EOF`.

```
char *fgets( char *s, int n, FILE *strm);
```

Einlesen einer Zeile (max. ***n*-1** Zeichen) aus Datei. Liefert `NULL` bei Dateiende oder Fehler.

7. Bibliotheken

Tests für Zeichenklassen `<ctype.h>`

Die folgenden Funktionen geben einen logischen Wert zurück, der entscheidet ob das Argument **`c`** zu einer bestimmten Klasse von Zeichen gehört.

`int isalnum( int c);` Buchstabe oder Ziffer
`int isalpha( int c);` Buchstabe
`int iscntrl( int c);` Steuerzeichen
`int isdigit( int c);` Ziffer
`int isgraph( int c);` sichtbares Zeichen
`int islower( int c);` Kleinbuchstabe
`int isspace( int c);` Leerzeichen, Tabulator, Newline, etc.
`int isupper( int c);` Großbuchstabe

7. Bibliotheken

Zeichenketten `<string.h>`

Zeichenketten in C sind Arrays vom Typ **`char`**. Eine Zeichenkette endet immer mit dem Symbol **`"\0"`**, auch wenn das Array länger ist.

`size_t strlen( char *s);`

Liefert die Länge von **`s`**.

`char *strcpy( char *s, char * t);`

Kopiert **`t`** nach **`s`**, liefert **`s`**. (Das Array für **`s`** muss hinreichend lang sein.)

`char *strcat( char *s, char *t);`

Fügt **`t`** an **`s`** an, liefert **`s`**. (Das Array für **`s`** muss hinreichend lang sein.)

`int strcmp( char *s, char *t);`

Vergleicht **`s`** mit **`t`**. Liefert **<0**, falls **`s<t`** (alphabetisch); **0**, falls **`s=t`**; **>0**, falls **`s>t`** .

7. Bibliotheken

Mathematische Funktionen `<math.h>`

Bei Verwendung von **`math.h`** muss die Compileroption **`-lm`** angegeben werden.

`M_PI (double)` π

`double sin( double x);` Sinus
`double cos( double x);` Cosinus
`double tan( double x);` Tangens
`double asin( double x);` Arcus Sinus
[-1, 1] $\longrightarrow$ [$-\pi/2, \pi/2$]
`double acos( double x);` Arcus Cosinus
[-1, 1] $\longrightarrow$ [0, π]
`double atan( double x);` Arcus Tangens
$\longrightarrow$ [$-\pi/2, \pi/2$]
`double sinh( double x);` Sinus Hyperbolicus
`double cosh( double x);` Cosinus Hyperbolicus
`double tanh( double x);` Tangens Hyperbolicus

7. Bibliotheken

Mathematische Funktionen `<math.h>`

`double exp( double x);` **exp(x)**

`double log( double x);` **ln(x)**

`double log10( double x);` **$\log_{10}$(x)**

`double pow( double x, double y);`

x^y. Fehler, wenn **x=0** und **y≤0** oder wenn **x<0** und **y** nicht ganzzahlig ist.

`double sqrt(double x);` Quadratwurzel, **x≥0**

`double ceil(double x);` kleinste Ganzzahl **≥x**

`double floor(double x);` größte Ganzzahl **≤x**

`double fabs(double x);` Absolutwert

`double fmod(double x, double y);`

Divisionsrest

7. Bibliotheken

Hilfsfunktionen `<stdlib.h>`

Umwandlung von Zeichenketten in Zahlen

```
double atof( char *s);
int atoi( char *s);
long atol( char *s);
```

Umwandlung der Zeichenkette **`s`** in eine Zahl vom Typ **`double`** / **`int`** / **`long`**.

Zufallszahlen

```
int rand();
```

Pseudo-Zufallszahl im Bereich **0** bis **`RAND_MAX`**.

```
void srand( int seed);
```

Startet den Zufallszahlengenerator mit einem neuen Wert. Um bei jedem Programmlauf eine neue Folge von Pseudozufallszahlen zu erhalten, empfiehlt es sich, für **`seed`** die Zeit einzusetzen (siehe Funktion **`time`** in **`<time.h>`**).

7. Bibliotheken

Hilfsfunktionen `<stdlib.h>`

Dynamische Speicherverwaltung in C

```
void *malloc( size_t n);
```
Reservieren eines Speicherblockes von **`n`** Bytes.

```
void *realloc( void *p, size_t n);
```
Kopiert den Inhalt des Speicherblockes auf den **`p`** zeigt in einen neuen Block der Größe **`n`** und liefert einen Zeiger auf diesen Block. Der ursprüngliche Block wird freigegeben.

```
void free( void *p);
```
Freigabe eines reservierten Speicherblockes. Die Adresse in **`p`** muss vorher dynamisch (z.B. mit malloc) reserviert worden sein oder **`p=NULL`** (kein Effekt).

Programmabbruch

```
void exit( int status);
```