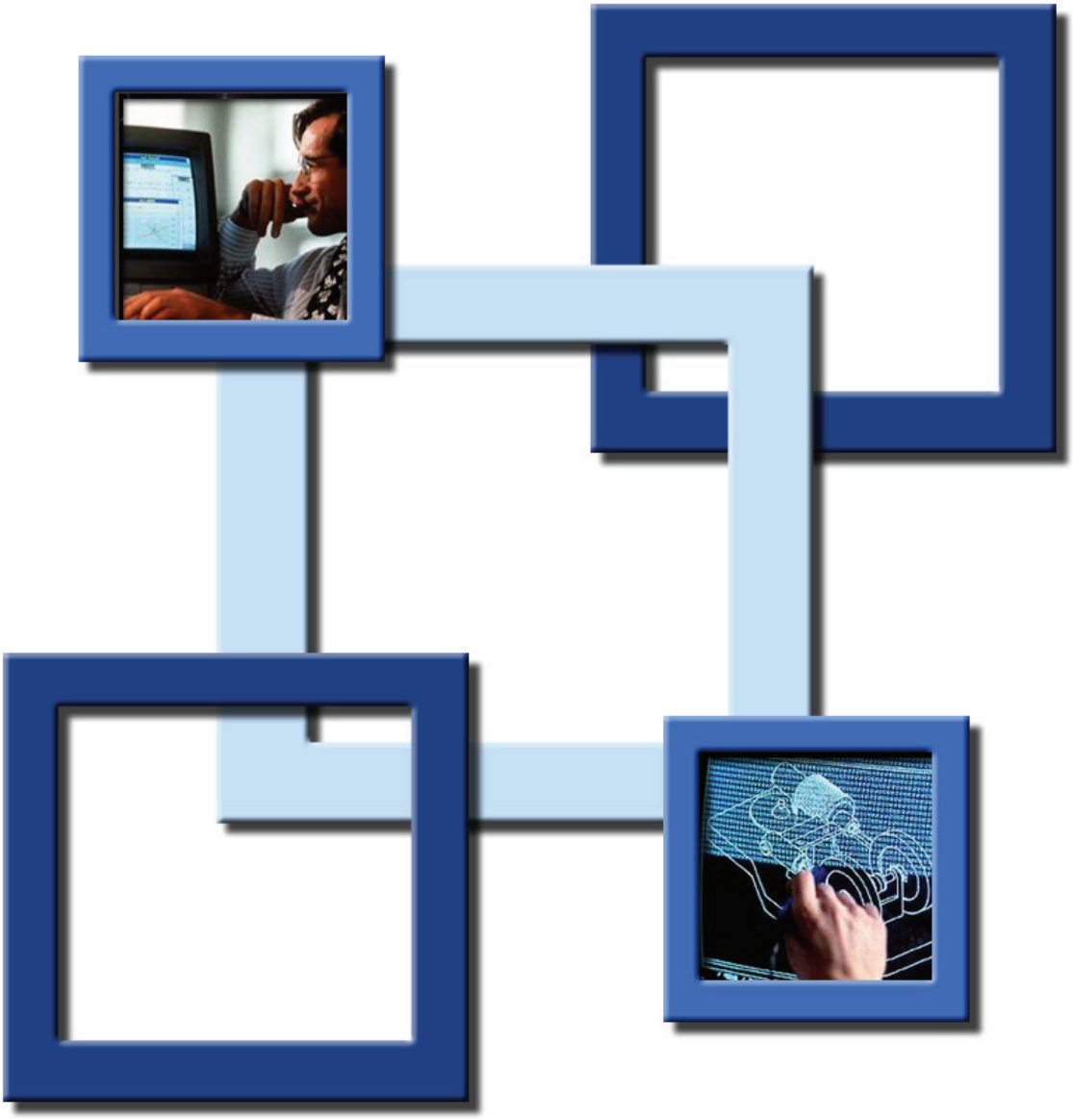


C++ Befehlssammlung

Formelsammlung

Inhaltsverzeichnis

1. Teile eines C++ Programms 5
2. Kommentare 5
3. Variablen 5
   Variablentypen 5
   Variablen definieren 6
   Gross - und Kleinschreibung 6
   typedef 6
   sizeof 7
4. Ein- und Ausgabe 7
   stdio.h 7
   iostream.h 9
5. Konstanten 10
   Konstanten mit #define 10
   Konstanten mit const 10
   Aufzählungskonstanten 10
6. Operatoren 11
   Mathematische Operatoren 11
   Zusammengefasste Operatoren 12
   Präfix und Postfix 12
7. if - Anweisung 13
8. while - Schleife 13
9. continue und break 13
10. do...while - Schleife 14
11. for - Schleife 14
    Mehrfache Initialisierung und Inkrementierung 14
    Leere for - Schleifen 15
12. Switch-Anweisung 15
13. Funktionen 16
    Deklaration 16
    Lokale Variablen 17
    Globale Variablen 17
    Rückgabewerte 17
    Standartparameter 17
    Funktionen überladen 18
14. Klassen 18
    Klassendeklaration 18
    Klassenmethoden implementieren 19

Private und Public 20
Konstruktor 20
Destruktor 21
Kopierkonstruktor 21
Inline - Implementierung 23
Überladene Elementfunktionen 23
15. Objekte 24
Objekte definieren 24
Auf Klassenelemente zugreifen 24
Objekte initialisieren 24
16. Zeiger 25
Indirektionsoperator 26
17. new und delete 26
new 26
delete 27
18. Objekte auf dem Heap 28
Erzeugen 28
Löschen 28
Auf Datenelemente zugreifen 28
19. Zeiger this 29
20. Konstante Zeiger 29
21. Referenzen 30
Referenzen erzeugen 30
Adressoperator bei Referenzen 30
Was kann man referenzieren? 31
Funktionsargumente als Referenz übergeben 31
Parameter mit Zeigern übergeben 31
Parameter mit Referenzen übergeben 32
22. Arrays 33
Über das Ende eines Array schreiben 33
Arrays initialisieren 34
Grösse des Arrays ermitteln 34
Arrays von Objekten 34
23. Mehrdimensionale Arrays 35
Mehrdimensionale Arrays initialisieren 35
24. Arrays von Zeigern 36
25. Arrays im Heap 36
26. char - Array 37
27. strcpy() und strncpy() 38
28. Vererbung 38

Syntax der Ableitung....39
Protected....40
Konstruktoren und Destruktoren....40
Funktionen redefinieren....40
Methoden der Basisklasse verbergen....41
Basismethoden aufrufen....41
29. Virtuelle Methoden:....42
Virtuelle Destruktoren....43
Datenteilung (Slicing)....44
30. Dynamische Typenumwandlung....45
31. Abstrakte Datentypen....47
32. Statische Datenelemente....49
33. Statische Elementfunktionen....51
34. Zeiger auf Funktionen....53
Deklaration....53
Kurzaufruf....54
Arrays mit Zeigern auf Funktionen....55
Zeiger auf Funktionen an andere Funktionen übergeben....56
typedef bei Zeigern auf Funktionen....57
35. Zeiger auf Elementfunktionen....59
Arrays von Zeigern auf Elementfunktionen....60
36. Präprozessor....61
#include....61
#define....61
#undef....62
#ifdef, #ifndef, #else, #endif....62
Schutz vor Mehrfachdeklaration....63
37. Makros....63
String-Manipulation....65
Vordefinierte Makros....66
assert....66
38. try - Blöcke....67

1. Teile eines C++ Programms

```
#include <stdio.h>     // Präprozessor Befehle

int main(void)         // Funktion main()
{
  Anweisungen;
  return(0);           // Rückgabewert
}
```

2. Kommentare

a) Alle Zeichen nach den doppelten Schrägstrichen bis Zeilenende werden ignoriert.

```
#include <stdio.h>     // Präprozessor Befehle
```

b) Der Compiler ignoriert alle Zeichen vom einleitenden "/*" bis zum abschliessenden "*/" Kommentarzeichen.

```
#include <stdio.h>     /* Präprozessor Befehle */

int main(void)
{
  /*Anweisungen;
  return(0);*/
}
```

3. Variablen

Variablentypen

Typ	Grösse	Wert
unsigned short int	2 Byte	0 bis 65'535
short int	2 Byte	-32'768 bis 32'767
unsigned long int	4 Byte	0 bis 4'294'967'295
long int	4 Byte	-2'147'483'648 bis 2'147'483'647
char	1 Byte	256 Zeichenwerte
float	4 Byte	1.2E-38 bis 3.4E38
double	8 Byte	2.2E-308 bis 1.8E308

Variablen definieren

```
main()
{
  unsigned short int breite;
```

a) Mehrere Variablen gleichzeitig erzeugen:

```
  ussigned short int breite, laenge, hoehe;
```

b) Werte Variablen zuweisen:

```
  unsigned short int breite;
  breite = 5;
```

c) Diese beiden Schritte lassen sich zusammenfassen:

```
  unsigned short int breite = 5;
```

d) Variablen in `for` - Schleifen deklarieren:

```
  for (int i = 0; i < 10; i++)
  {
    Anweisungen;
  }
}
```

Die Variable `i` ist nur innerhalb der `for` - Schleife gültig.

Gross - und Kleinschreibung

C++ beachtet die Gross und Kleinschreibung. Eine Variable namens `alter` unterscheidet sich also von `Alter` und diese wiederum von `ALTER`.

typedef

Mit dem Schlüsselwort `typedef` kann man einen Alias für eine Typen-definition erstellen:

```
typedef unsigned short int USHORT;    //mit typedef definiert

int main(void)
{
  USHORT width = 5;
  ...
```

sizeof

Die Funktion `sizeof` gibt die Grösse des als Parameter übergebenen Objekts in Byte zurück:

```
sizeof(int);
```

So erhalten wir hier den Wert 2, da ein Integer 2 Byte gross ist.

4. Ein- und Ausgabe

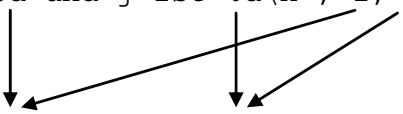
stdio.h

Die C - Funktionen `printf` und `scanf` von `stdio.h` können in C++ verwendet werden. Allerdings sollte man `cin` und `cout` den alten Funktionen `printf` und `scanf` vorziehen.

printf

Die Funktion `printf` dient zur formatierten Ausgabe von Variablen und Konstanten:

```
int i = 1;
int j = 2;
printf("i ist %d und j ist %d\n", i, j);
```



Ausgabe: `i ist 1 und j ist 2`

Ein Prozentzeichen (`%`) leitet jeweils eine Formatanweisung ein.
Im Beispiel haben wir `%d`, was für `int` in Dezimaldarstellung steht. Also wird `i` und später auch `j` in diesem Format ausgegeben.

Format	Typisierung der Parameter u. Darstellung	Beispiel Zahl 78
%d	int, short int oder char als Zahl in Dezimalnotation	78
%c	int, short int oder char als Zeichen	N
%x	int, short int, oder char als Zahl in Hexadezimalnotation mit kleinen Buchstaben a...f	4e

%X	int, short int, oder char als Zahl in Hexadezimalnotation mit grossen Buchstaben A...F	4E
%o	int, shortint oder char als Zahl in Oktalnotation	116
%u	unsigned int, unsigned short int oder unsigned char als Zahl in Dezimalnotation	78
%f	float in Fliesskommaschreibweise	78.000000
%e	float in Exponentialschreibweise mit kleinem e	7.800000e+001
%E	float in Exponentialschreibweise mit grossem E	7.800000E+001
%g	float in Exponentialschreibweise oder Fliesskommaschreibweise ohne ausgabe der nullen	
%s	char (bzw. char[ ]) zur Ausgabe von Zeichenketten	Numerische Werte (78) können nicht ausgegeben werden
%%	Diese Zeichenfolge ist keine Formatanweisung, sondern dient zur Ausgabe des Prozentzeichens	

Folgende Beispiele zeigen, wie die Anzahl Stellen, vor und hinter dem Komma formatiert werden können:

```
printf("Wert 1 = %5d", x);
```

`x` wird im Format int mit 5 Stellen ausgegeben.

```
printf("Wert 2 = %5.2f", y);
```

`y` wird im Format float mit 5 Stellen vor und 2 Stellen nach dem Komma.

scanf

`scanf` dient zur Eingabe von Daten. Da der Wert als Zeiger übergeben wird muss man der einzulesenden Variablen ein Adressoperator (`&`) (siehe Kap. 16) voranstellen:

```
int value;
scanf("%d", &value);
```

Wie bei der Funktion printf leitet ein Prozentzeichen (`%`) jeweils eine Formatanweisung ein (siehe Kap. 4 / printf).

iostream.h

Die Befehle von iostream.h entsprechen der C++ Philosophie. Das heisst sie sind typensicher und erweiterbar. Man muss sich nicht darum kümmern, welchen Typ (int, char, long...) das einzulesende bzw. auszugebende Objekt hat, der Compiler nimmt diese Arbeit ab.

cout

Das Objekt `cout` dient zur Ausgabe. In Anschluss an `cout` steht der Umleitungsoperator (<<). Alles was auf den Umleitungsoperator folgt, wird ausgegeben:

```
cout << "Hello World";
```

Will man mehrere verschiedene Variablen und Zeichenfolgen gleichzeitig ausgeben schreibt man jeweils zischen die Ausgabeelemente einen Umleitungsoperator:

```
int value = 10
float height = 2.5;

cout << "Wert: " << value << "Höhe: " << height;
```

Neben dem Ausgabestream cout sind in C++ noch weitere standart Ausgabestreams definiert. Dies sind cerr für Fehlermeldungen und clog für Portokolllierungen.

cin

Mit dem Objekt `cin` lassen sie Daten einlesen. Der Umleitungsoperator zeigt folglich in die andere Richtung (>>):

```
int value;
cin >> value;
```

5. Konstanten

Konstanten mit #define

```
#define stunden 15
```

Wichtig ist, dass `stunden` keinen besonderen Typ (z.B. int oder char) aufweist. Der Präprozessor nimmt eine einfache Textersetzung vor und schreibt überall, wo der String `stunden` erscheint, 15 in den Quelltext.

Konstanten mit const

In C++ gibt es eine neue, bessere Lösung zur Definition von Konstanten:

```
const unsigned short int stunden = 15;
```

Dieses Beispiel definiert ebenfalls Konstante namens `stunden`, mit dem Unterschied, dass diesmal `stunden` als Typ `unsigned short int` definiert ist.

Aufzähungskonstanten

Aufzählungskonstanten erzeugen einen Satz von Konstanten mit einem Bereich von Werten:

```
enum Farbe {ROT, BLAU, GRUEN, WEISS, SCHWARZ};
```

- `Farbe` erhält den Namen einer Aufzählung, d.h., einen neuen Typ.
- `ROT` wird zu einer symbolischen Konstanten mit dem Wert 0, `BLAU` wird zu einer Konstanten mit dem Wert 1, `GRUEN` hat den Wert 2 usw.

Jede Konstante lässt sich mit einem Wert initialisieren:

```
enum Farbe {ROT=100, BLAU=500, GRUEN, WEISS, SCHWARZ=700};
```

Die Konstanten enthalten danach folgende Werte:

```
ROT     = 100  WEISS   = 501
BLAU    = 101  SCHWARZ = 700
GRUEN   = 500
```

6. Operatoren

Operatoren mit höherem Vorrang werden zuerst ausgewertet. In der Tabelle gilt: Je niedriger der Rang, desto höher der Vorrang:

Rang	Bezeichnung	Operator
1	Zugriffsoperator	`::`

2	Elementauswahl, Indizierung, Funktionsaufrufe, Inkrement und Dekrement in Postfix-Notation	`. ->`
		`()`
		`++ --`

3	`sizeof`, Inkrement und Dekrement in Präfix-Notation, `AND`, `NOT`, unäres Minus und Plus, Adressoperator und Dereferenz, `new`, `new[]`, `delete`, `delet[]`, Typenumwandlung	`++ --`
		`^ !`
		`- +`
		`& *ü`

4	Elementauswahl für Zeiger	`.* ->*`
5	Multiplikation, Division, Modulo	`* / %`
6	Addition, Subtraktion	`+ -`
7	Verschiebung	`<< >>`
8	Vergleichsoperatoren	`< <= > >=`
9	Gleich, Ungleich	`== !=`
10	Bit-weises `AND`	`&`
11	Bit-weises Exklusiv-`OR`	`^`
12	Bit-weises `OR`	`|`
13	Logisches `AND`	`&&`
14	Logisches `OR`	`||`
15	Bedingung	`?:`

16	Zuweisungsoperatoren	`= *= /= %=`
		`+= -= <<=`
		`>>= |= ^=`

17	`Throw`-Operator	`throw`
18	Komma	`,`

Mathematische Operatoren

Die Grundrechenarten Addition und Subtraktion arbeiten wie gewohnt.

Setzt man bei der Division im Zähler eine Ganzzahl ein, wird eine Ganzzahldivision durchgeführt:

`x = 5/2` → x = 2

Will man als Ergebnis aber eine reelle Zahl muss man eine gebrochene Zahl im Zähler einsetzen:

```
x = 5.0/2
```
→ x = 2.5

Der Modulo - Operator (%) gibt den Rest einer Ganzzahldivision zurück:

```
x = 21%4
```
→ x = 1 da 21 / 4 = 5 Rest 1

Zusammengefasste Operatoren

Normal	Zusammengefasst
`counter = counter + 1;`	`counter += 1;`
`counter = counter - 1;`	`counter -= 1;`
`counter = counter * 1;`	`counter *= 1;`
`counter = counter / 1;`	`counter /= 1;`
`counter = counter % 1;`	`counter %= 1;`
usw.	

Präfix und Postfix

Der Inkrementoperator und der Dekrementoperator existieren in zwei Spielarten: Präfix und Postfix.

Präfix

```
++ counter      -- counter
```

Inkrementiere den Wert zuerst und übertrage in dann.

Postfix

```
counter ++      counter --
```

Übertrage den Wert zuerst und inkrementiere dann.

Beispiel

```
x = 5;
```

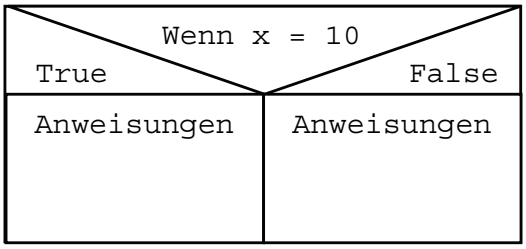
```
a = ++ x;
```
→ a = 6; x = 6

```
x = 5;
a = x ++;
```
→ a = 5; x = 6

7. if - Anweisung

Hat man nur eine Anweisung kann man die Klammern weggelassen.

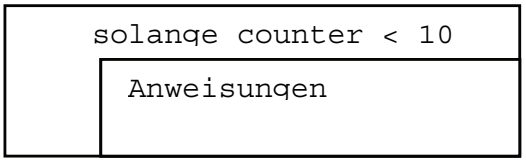
```
if (x == 10)
{
  Anweisungen;
}
else
{
  Anweisungen;
}
```



8. while - Schleife

Hat man nur eine Anweisung kann man die Klammern weggelassen.

```
while (conter < 10)
{
  Anweisungen:
}
```



Verwendet man für die zu testende Bedingung die Zahl 1 erhält man eine Endlosschleife, da 1 immer True ist.

```
while (1)                          // Endlosschleife
{
  Anweisungen:
}
```

9. continue und break

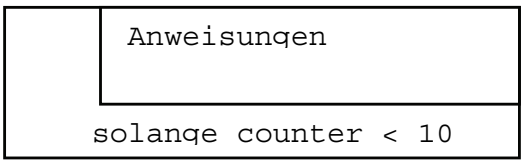
Will man in einer while - Schleife an den Anfang zurückkehren bevor die gesamte Gruppe von Anweisungen in der Schleife abgearbeitet sind, benutzt man die `continue` - Anweisung, die einen Sprung an den Beginn der Schleife bewirkt.

Die Anweisung `break` führt zum unmittelbaren Austritt aus der Schleife, ohne dass die Ende - Bedingungen erfüllt sind.
Diese Anweisung wird unteranderem auch benutzt um aus Endlosschleifen (`while(1)/while(true)`) auszutreten.

10. do...while - Schleife

Die `do...while` - Schleife führt den Rumpf der Schleife aus, bevor getestet wird. Somit ist gesichert das der Rumpf mindestens einmal abgearbeitet wird.

```
do
{
  Anweisungen;
} while (counter < 10)
```



11. for - Schleife

Eine `for` - Schleife fasst die drei Schritte Initialisierung, Test und Inkrementierung in einer Anweisung zusammen:

```
for (counter = 0; counter < 5; counter++)
{
  Anweisungen;
}
```

Die Variable kann auch gleich in der Schleife deklariert werden. Die so deklarierte Variable ist dann allerdings nur innerhalb der Schleife gültig:

```
for (int counter = 0; counter < 5; counter++)
{
  Anweisungen;
}
```

Mehrfache Initialisierung und Inkrementierung

Man kann mehrere Variablen auf einmal initialisieren, einen zusammengesetzten logischen Ausdruck testen und mehrere Anweisungen ausführen. Die Anweiungen werden jeweils zur ein Komma getrennt:

```
for (i = 0, j = 0; i < 3; i++, j++)
{
  Anweisungen;
}
```

Leere for - Schleifen

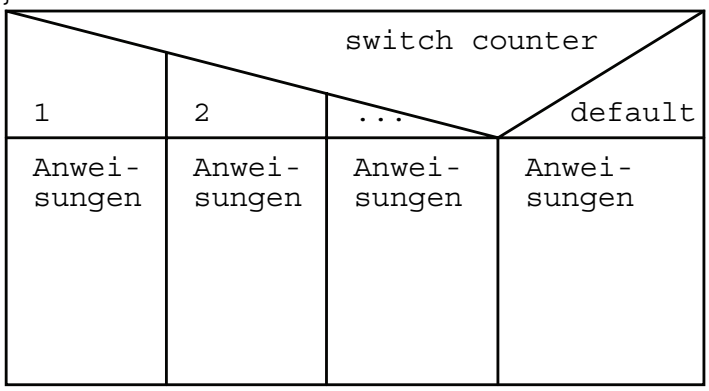
Da man auch Anweisungen im Kopf von `for` - Schleifen verpacken kann, kann manchmal auf den Anweisungsrumpf verzichtet werden. Dann schliesst man den Kopf mit einem Semikolon ab:

```
for (int i = 0; i < 5; cout << "i: " << i++ <<endl);
```

12. Switch-Anweisung

Eine Alternative zur Verschachtelung vieler `if` - Anweisungen bietet die `switch` - Anweisung:

```
switch (counter)        // switch Variable counter
{
  case 1:               // wenn Variable counter = 1
  {
    Anweisungen;
    break;
  }
  case 2:               // wenn Variable counter = 2
  {
    Anweisungen;
    break;
  }
  ...                   // und so weiter

  default;              // sonst...
  {
    Anweisungen;
  }
}
```



Wenn einer der `case` - Werte mit dem Ausdruck übereinstimmt, springt die Ausführung zu diesen Anweisungen und arbeitet diese bis zum Ende des `switch` - Blocks oder bis zu nächsten `break` - Anweisung ab.

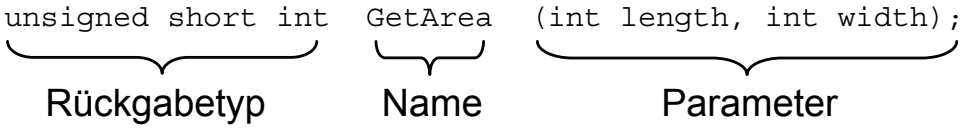
Gibt es keine Übereinstimmung, verzweigt die Ausführung zu der optionalen `default` - Anweisung.

Der Nachteil der `switch` - Anweisung ist, dass `case` - Werte keine Variablen und Konstanten (ausser Konstanten mit `#define`) sein dürfen.

13. Funktionen

Deklaration

Die Parameterliste führt - durch Komma getrennt - alle Parameter zusammen mit deren Typen auf.



Der Prototyp und die Definition einer Funktion müssen hinsichtlich Rückgabetyp, Name und Parameterliste genau übereinstimmen.

Beispiel

```
#include <iostream.h>

// Funktionsprototyp

int GetArea(int length, int width);
int main(void)
{
  int area, inlength, inwidth;
  cout << "Laenge eingeben: ";
  cin >> inlaenge;
  cout << "Breite eingeben: ";
  cin >> inbreite;
  area = GetArea(inlength, inwidth);

  cout << "Flaeche = " << area;
}


// Funktionsdefinition

int GetArea(int l, int w)
{
  return(l * w);
}
```

Lokale Variablen

Variablen die innerhalb einer Funktion deklariert werden existieren nur innerhalb der Funktion. Kehrt man aus der Funktion zurück sind lokale Variablen nicht mehr zugänglich.

Parameter sind auch lokale Variablen, da die Funktion von jedem zu übergebenden Argument eine lokale Kopie erstellt.

Globale Variablen

Variablen die ausserhalb aller Funktionen definiert sind (globale Variablen), sind für alle Funktionen im Programm einschliesslich der Funktion `main` verfügbar.

Rückgabewerte

Funktionen geben einen Wert oder `void` zurück. `void` ist ein Signal an den Compiler, dass kein Wert zurückgeben wird.

Die Rückgabe eines Wertes erfolgt mit dem Schlüsselwort `return`

```
return 5;

return(x > 5);
return(MyFunction());
```

Der Wert der zweiten Anweisung `return(x > 5);` ist 0, wenn `x` nicht grösser als 5 ist. Andernfalls lautet der Rückgabewert 1. Es wird der Wert des Ausdrucks (0 = False; 1 = True) zurückgeben und nicht `x`.

Standartparameter

Der Prototyp der Funktion deklariert für den Parameter einen Standartwert, der beim Funktionsaufruf verwendet wird, wenn kein Wert übergeben wird.

```
long myFunction(int x = 50)   // Funktionsprototyp
```

Die Funktionsdefinition ändert sich bei der Deklaration eines Standartparameters nicht:

```
long myFunction(int x)        // Funktionsdefinition
```

Funktionen überladen

In C++ können mehrere Funktionen mit demselben Namen erzeugt werden. Sie müssen sich lediglich in ihrer Parameterliste unterscheiden. Man bezeichnet dies als Überladen von Funktionen:

```
int myFunction(int, int);
int myFunction(long, long);
int myFunction(long);
```

14. Klassen

Durch die Deklaration einer Klasse erzeugt man einen neuen Typ. Eine Klasse ist eine Sammlung von Variablen - häufig mit unterschiedlichen Typen - kombiniert mit einer Gruppe verwandter Funktionen.

Klassendeklaration

Normalerweise wird die Klassendeklaration separat in einem h - File implementiert.
Man nennt die Deklaration der Klasse auch Schnittstelle (Interface), da der Benutzer daraus alle Informationen zum Arbeiten mit der Klasse entnehmen kann. Von der Implementation der Klassenmethoden braucht er nichts zu wissen.

```
// Deklaration der Klasse Car (car.h)

#ifndef _Car_H
#define _Car_H

class Car
{
  public:                     // Öffentliche Deklarationen
    Car();                    // Standartkonstruktor
    Car(int y);               // Konstruktor
    ~Car();                   // Destruktor
    void Drive();
    void Stop();
    void SetYear(int y);
```

```
    int GetYear();
    int GetMaxVelocity();

  private:                    // Private Deklerationen
    int year
};

#endif
```

Klassenmethoden implementieren

Die Klassen Methoden werden in einem separaten cpp - File implementiert. Dabei muss die Headerdatei mit der dazugehörenden Klassendeklaration mit `#include "Car.h"` eingebunden werden.

```
// Implementation der Klasse Car (car.cpp)

#include "car.h"              // h-File mit Klassendeklaration
#include <iostream.h>

Car::Car                      // Standartkonstruktor
{
  year = 1991;
}

Car::Car(int y)               // Konstruktor
{
  year = y;
}

Car::~Car()                   // Destruktor
{
}

void Car::Drive()
{
  cout << "Car is driving...";
}

void Car::Stop()
{
  cout << "Car stoped";
}

void Car::SetYear(int y)
{
  year = y;
}

int Car::GetYear()
{
  return(year);
}
```

```
int GetMaxVelocity()
{
  return(200);
}
```

Private und Public

Daten und Methoden einer Klasse sind standardmässig privat. Das heisst, dass man auf diese Elemente nur innerhalb der Methoden der Klasse zugreifen kann. Gegen Zugriffe von ausserhalb des Objekts sind sie geschützt.

Mit den Schlüsselwörtern `public` und `private` lässt sich bestimmen, welche Elemente der Klasse privat und welche
öffentlich, also von ausserhalb des Objekts zugänglich, sein sollen.

```
class Car
{
  public:                    // Öffentliche Deklarationen
    ...

  private:                   // Private Deklarationen
    ...
};
```

Konstruktor

Klassen haben eine spezielle Elementfunktion, einen sogenannten Konstruktor, um die Datenelemente einer Klasse zu initialisieren.
Der Konstruktor ist eine Klassenmethode mit dem selben Namen wie die Klasse:

```
class Car
{
  public:
    Car();                   // Standartkonstruktor
    Car(int y);              // Konstruktor
    ...
```

Ein Konstruktor ohne Parameter heisst Standartkonstruktor und wird vom Compiler automatisch erzeugt, wenn keine Konstruktoren deklariert wurden.
Der vom Compiler bereitgestellte Standartkonstruktor führt keine Aktion aus.

Es können mehrere Konstruktoren mit demselben Namen erzeugt werden. Sie müssen sich lediglich in ihrer Parameterliste unterscheiden. Man spricht von Überladen

Destruktor

Zur Deklaration eines Konstruktors gehört normalerweise auch die Deklaration eines Destruktors.
Genau wie Konstruktoren Objekte der Klasse erzeugen und initialisieren, führen Destruktoren Aufräumungsarbeiten nach Zerstörung des Objekts aus und geben reservierten Speicher frei.
Ein Destruktor hat immer den Namen der Klasse, wobei eine Tilde (~) vorangestellt ist:

```
class Car
{
  public:
    ...
    ~Car();                    // Destruktor
    ...
```

Destruktoren übernehmen keine Argumente und haben auch keinen Rückgabewert.

Kopierkonstruktor

Der Standartkopierkonstruktor wird immer dann aufgerufen wenn eine Kopie eines Objekts erzeugt wird.
Alle Kopierkonstruktoren übernehmen einen Parameter, eine Referenz auf ein Objekt der selben Klasse.

```
Car(const Car & theCar);
```

Hier übernimmt der `Car` - Konstruktor eine konstante Referenz auf ein existierendes `Car` - Objekt. Ziel des Kopierkonstruktors ist das Anlegen einer Kopie von `theCar`.
Der Standartkopierkonstruktor legt eine sogenannte Flache Kopie an. Das heisst er kopiert einfach alle Elementvariablen des Originals.

Dies kann aber gefährlich sein, wenn man z.B. Elementvariablen hat, die Zeiger (Kap. 15) auf Objekte im Heap (Kap. 16) sind. Dann zeigen

Kopie und Original an die selbe Stelle und wenn z.B. das Original gelöscht wird, gibt dieses den Speicher im Destruktor frei währendem die Kopie immer noch darauf zugreift!
Deshalb muss man in solchen Fällen einen eigenen Kopierkonstruktor anlegen:

```
01: class Car
02: {
03:   public:
04:     Car();                              // Standartkonstruktor
05:     Car(const Car &);                   // Kopierkonstruktor
06:     ~Car();                             // Destruktor
07:     int GetYear(); {return *year}       // Inline-Implementierung
08:     ...
09:   private:
10:    int *year;
11: }
12: Car::Car()
13: {
14:   year = new int;
15:   *year = 1990;
16: }
17: Car::Car(const Car & rhs)
18: {
19:   year = new int;
20:   *year = rhs.GetYear();
21: }
22: Car::~Car()
23: {
24:   delete year;
25:   year = 0;
26: }
```

Der Kopierkonstruktor beginnt in Zeile 17. Der Parameter ist wie bei einem Kopierkonstruktor üblich mit `rhs` benannt, was für right-hand-side - zur rechten Seite - steht.

- Zeile 19 reserviert speicher auf dem Heap (Kap. 16)
- Zeile 20 der Wert aus dem existierenden `Car` - Objekt wird in die neue Speicherstelleübertragen.

Der `rhs` Parameter ist ein `Car` - Objekt, dessen Übergabe an den Kopierkonstruktor als konstante Referenz erfolgt.
Das neue `Car` - Objekt kann direkt auf seine eigenen Elementvariablen verweisen, muss aber die Elementvariablen von `rhs` mit den öffentlichen Zugriffsmethoden ansprechen.

Inline - Implementierung

Klassenmethoden können inline implementiert werden. Das heisst die Implementierung findet unmittelbar in der Deklaration statt.
Dazu gibt es zwei Varianten:

```
class Car
{
  public:                                  // Öffentliche Deklarationen
    inline int Car::GetYear()
    {
      return(year);
    }
};
```

Man kann auch die Definition einer Funktion in die Deklaration der Klasse schreiben, was die Funktion automatisch zu einer Inline - Funktion macht:

```
class Car
{
  public:                                  // Öffentliche Deklarationen
    int GetYear()
    {
      return(year);                        // inline
    }
};
```

Überladene Elementfunktionen

Elementfunktionen lassen sich genau gleich wie normale Funktionen überladen (Polymorphie). Also Elementfunktionen mit dem selben Namen aber unterschiedlicher Parameterliste.

```
class Rectangle
{
  public:
    ...
    void DrawShape();
    void DrawShape(int aWidth, int aHeight);
    ...
```

15. Objekte

Objekte definieren

Ein Objekt des neuen Typs definiert man fast genauso wie eine Integervariable:

```
unsigned int Weight;                    // eine Integer-Variable def.
Car Mercedes;                           // ein Car-Objekt definieren
```

`Mercedes` ist ein Objekt vom Typ `Car`, wie `Weight` eine Variable vom Typ `unsigned int` ist.

Auf Klassenelemente zugreifen

Nachdem man ein Objekt definiert hat, kann man mit dem Punktoperator (`.`) auf die Elemente des Objekts zugreifen:

Man kann einen Wert an eine Elementvariable zuweisen.

```
Mercedes.year = 1990;
```

In gleicher Weise ruft man eine Funktion auf.

```
Mercedes.Drive();
Mercedes.SetYear(1990);
```

Objekte initialisieren

Elementvariablen von Objekten lassen sich im Rumpf oder im Initialisierungsteil des Konstruktors initialisieren:

Im Rumpf

```
Car::Car                            // Standartkonstruktor
{
  year = 1991;                      // Rumpf des Konstruktors
  petrol = 10;
}
```

Im Initialisierungsteil

```
Car::Car():                    // Standartkonstruktor
year(1991),                    // Initialisieurngsliste
petrol(10)
{
  Anweisungen;                 // Rumpf des Konstruktors
}
```

Referenzen (Kap. 25) und Konstanten (Kap. 4) müssen initialisiert werden, da sie keine Zuweisung erlauben. Verwendet man also Referenzen oder Konstanten als Datenelemente müssen diese immer im Initialisierungsteil initialisiert werden.

16. Zeiger

Ein Zeiger ist eine Variable, die eine Speicheradresse enthält.

```
unsigned int howOld = 50;       // eine Integer-Variabel erzeugen
unsigned int *pAge = 0;         // einen Zeiger erzeugen
pAge = &howOld;                 // die Adresse von howOld pAge
                                // zuweisen
```

Die erste Zeile erzeugt die Variable `howOld` mit dem Typ `unsigned int` und initialisiert sie mit dem Wert 50.

In er zweiten Zeile wird `pAge` als Zeiger auf den Typ `unsigned int` deklariert und mit 0 initialisiert.
Wird ein Zeiger mit 0 initialisiert spricht man von einem Nullzeiger. Zeiger sollten bei ihrer Deklaration immer initialisiert werden. Wenn man nicht weiss, was man dem Zeiger zuweisen soll, wählt man einfach den Wert 0. Sonst entstehen sogenannte wilde Zeiger die irgendwo in den Speicher zeigen.
Zeile drei zeigt wie die Adresse von `howOld` dem Zeiger `pAge` zugewiesen wird. Die Zuweisung einer Adresse kennzeichnet der Adressoperator `(&)`. Ohne ihn hätte man den Wert von `howOld` zugewiesen. Das ganze lässt sich um eine Schritt kürzen:

```
unsigned int howOld = 50;       // eine Integer-Variable
                                // erzeugen
unsigned int *pAge = &howOld;   // einen Zeiger auf howOld
                                // erzeugen
```

Indirektionsoperator

Der Indirektionsoperator (`*`) bezeichnet man auch als Dereferenzierungsoperator. Wenn ein Zeiger dereferenziert wird, ruft man den Wert an der im Zeiger gespeicherten Adresse ab.

Demnach geschieht in den Folgenden Beispielen beide Male das Selbe: Der Wert von `howOld` wird der Variable `yourAge` zugewiesen.

Beispiel 1

```
unsigned int yourAge;
yourAge = howOld;
```

Beispiel 2

```
unsigned int yourAge;
yourAge = *pAge;
```

Der Zeiger bietet indirekten Zugriff auf den Wert der Variablen, deren Adresse er speichert.

17. new und delete

Kehrt man aus einer Funktion zurück werden die lokalen Variablen, die sich auf dem Stack befinden, verworfen. Verwendet man globale Variablen um dieses Problem zu umgehen, hat man den Nachteil, dass fremde Funktionen die Daten in unerwarteter und nicht vorhergesehener Weise verändern können.
Um diese Nachteile zu umgehen speichert man oft Daten auf dem Heap, dem sogenannten Freispeicher, ab.

new

Mit dem Schlüsselwort `new` wird Speicher im Heap reserviert.
Der Rückgabewert von `new` ist eine Speicheradresse. Diese muss einem Zeiger zugewiesen werden.
Um einen `unsigned short int` im Heap zu erzeugen, geht man wie folgt vor:

```
unsigned short int *pPointer;
pPointer = new unsigned short int;
```

Man kann gleichzeitig auch den Zeiger initialisieren:

```
unsigned short int *pPointer = new unsigned short int;
```

Nun kann man im Bereich im Heap, auf den `pPointer` zeigt, Werte ablegen:

```
*pPointer = 72;
```

Wenn new keinen Speicher im Heap erzeugen kann, erhält man als Rückgabewert einen Nullzeiger. Man sollte also bei jeder neuen Anforderung von Speicher auf einen Nullzeiger prüfen.

delete

Da der Zeiger eine lokale Variable ist, geht er bei der Rückkehr aus der Funktion verloren. Aber der mit dem Operator `new` reservierte Speicher auf den der Zeiger zeigte wird nicht automatisch freigegeben und bleibt somit bis zum Programmende unzugänglich. Es entstehen Speicherlücken.

Man muss vor dem Verlassen der Funktion den reservierten Speicher mit dem Schlüsselwort `delete` an den Heap zurückgeben:

```
delete pPointer;
```

Um Speicherlücken zu vermeiden muss man zuerst den vom Zeiger referenzierte Speicher freigeben, bevor man den Zeiger neu zuweist: Speicherbereich mit dem Wert 72 wird nicht freigeben und wird unzugänglich:

```
unsigned short int *pPointer = new unsigned short int;
*pPointer = 72;
pPointer = new unsigned short int;
*pPointer = 84;
```

Speicherbereich mit dem Wert 72 wird freigeben:

```
unsigned short int *pPointer = new unsigned short int;
*pPointer = 72;
delete pPointer;
pPointer = new unsigned short int;
*pPointer = 84;
```

Achtung

Ruft man `delete` zweimal auf einen Zeiger auf, stürzt das Programm ab. Man sollte deshalb den Zeiger beim löschen auf 0 setzen, da ein `delete` auf einen Nullzeiger problemlos ausgeführt werden kann.

```
unsigned short int *pPointer = new unsigned short int;

delete pPointer;              // gibt Speicher frei
pPointer = 0;                 // setzt den Zeiger auf Null
...
delete pPointer;              // harmlos
```

18. Objekte auf dem Heap

Erzeugen

Wenn man ein Objekt vom Typ `Car` deklariert hat, kann man einen Zeiger auf diese Klasse deklarieren und ein `Car` - Objekt auf dem Heap instantiiren:

```
Car *pCar = new Car;
```

Damit wird der Standartkonstruktor (Konstruktor ohne Parameter), aufgerufen. Dies geschieht immer beim Erzeugen eines Objekts auf dem Stack oder dem Heap.

Löschen

Beim Aufruf von delete auf einen Zeiger zu einem Objekt auf dem Heap wird der Destruktor (zum erledigen von Aufräumarbeiten) des Objekts aufgerufen, bevor die Freigabe des Speichers erfolgt.

```
delete pCar;
```

Auf Datenelemente zugreifen

Auf Datenelemente und Funktionen von lokal erzeugten Objekten greift man mit dem Punktoperator (.) zu.
Um ein Objekt im Heap anzusprechen, muss man den Zeiger dereferenzieren und den Punktoperator auf dem Objekt aufrufen auf das der Zeiger verweist:

```
(*pCar).Drive();
```

Mit den Klammern stellt man sicher, dass `pCar` vor dem Zugriff auf `Drive` dereferenziert wird.

In C++ gibt es einen handlicheren Kurzoperator für den indirekten Zugriff, den Elementverweis - Operator (`->`):

```
pCar->Drive();
```

19. Zeiger this

Jede Elementfunktion einer Klasse verfügt über einen versteckten Parameter: den Zeiger `this`, der auf das eignen Objekt zeigt.

Die Aufgabe des Zeigers `this` besteht darin, auf das jeweilige Objekt zu verweisen, dessen Methode aufgerufen wurde.

20. Konstante Zeiger

Bei Zeigern kann man das Schlüsselwort `const` vor oder nach dem Typ (oder an beiden Stellen) verwenden:

```
const int *pOne;
int *const pTwo;
const int *const pThree;
```

- `pOne` ist ein Zeiger auf eine konstante Ganzzahl. Der Wert, auf den er zeigt, lässt sich nicht über diesen Zeiger ändern.

```
*pOne = 5;              // Compiler reagiert mit Fehler
```

- `pTwo` ist ein konstanter Zeiger auf eine Ganzzahl. Die Ganzzahl kann man ändern, aber `pTwo` kann nicht auf etwas anderes zeigen.

```
pTwo = &x;              // Compiler reagiert mit Fehler
```

- `pThree` ist ein konstanter Zeiger auf eine konstante Ganzzahl.

Deklariert man Zeiger als Konstant oder Zeiger auf Konstante erhält man bei Fehlern die durch nicht beabsichtigten Einsatz von Zeigern entstehen, Unterstützung durch den Compiler

Will man zum Beispiel, das ein Objekt nicht verändert werden kann, erzeugt man einen Zeiger auf ein konstantes Objekt:

```
const Car *pCar = new Car;
```

Nun kann man mit diesem Zeiger nur konstante Methoden des Objekts aufrufen. Versuchen wir das Objekt zu ändern erhalten wir eine Fehlermeldung des Compilers:

```
pCar->SetYear(1991);  // Compiler reagiert mit Fehler
```

21. Referenzen

Eine Referenz ist ein Alias - Name. Wenn man eine Referenz erzeugt, initialisiert man sie mit dem Namen eines anderen Objekts, dem Ziel. Von diesem Moment an wirkt die Referenz als alternativer Name für das Ziel und alles, was man mit der Referenz anstellt, bezieht sich tatsächlich auf das Ziel.

Referenzen erzeugen

Für die Integer - Variable `someInt` soll eine Referenz erzeugt werden. Die Deklaration einer Referenz besteht aus dem Typ des Zielobjekts, gefolgt vom Referenzoperator (`&`) und dem Namen der Referenz:

```
int &rSomeRef = someInt;
```

Referenzen können nicht erneut zugewiesen werden. Sie sind immer Aliase für ihr erstes Ziel. Macht man dennoch eine Neuzuweisung übergibt man dem Ziel einen Wert, anstatt ein neues Ziel zuzuweisen. C++ verwendet für den Referenzoperator (`&`) dasselbe Symbol wie für den Adressoperator.

Adressoperator bei Referenzen

Wenn man die Adresse einer Referenz abfragt, erhält man die Adresse des Ziels der Referenz.

Was kann man referenzieren?

Alle Objekte (mit benutzerdefinierten) lassen sich referenzieren. Zu beachten ist, dass man eine Referenz nicht auf eine Klasse erzeugt sondern auf ein Objekt:

```
int &rIntRef = int;           // falsch
```

Man muss `rIntRef` auf eine bestimmte Integer - Zahl initialisieren:

```
int howBig = 200;
int &rIntRef = howBig;
```

Ebenso initialisiert man eine Referenz nicht auf die Klasse `Car`:

```
Car &CarRef = Car;            // falsch

Car Mercedes;                 // richtig
Car &rCarRef = Mercedes;
```

Wenn man Zeiger löscht oder nicht initialisiert , sollte man ihnen Null zuweisen. Dies gilt für Referenzen nicht, da eine Referenz nicht Null sein darf.

Funktionsargumente als Referenz übergeben

Funktionen haben zwei Einschränkungen: Die Übergabe von Argumenten erfolgt als Wert und mit Hilfe der return - Anweisung kann nur ein einziger Wert zurückgegeben werden. Man kann die Einschränkungen umgehen, wenn man die Werte als Referenz, realisiert mit Zeigern oder Referenzen, an die Funktion übergibt.

Parameter mit Zeigern übergeben

Die Übergabe eines Zeigers bedeutet das man die Adresse des Objekts übergibt. Die Funktion kann dann den Wert an dieser Adresse beeinflussen:

```
void swap(int *x, int *y);

int main(void)
{
  int x = 5, x = 10;
  swap(&x, &y);
```

```
  return(0);
}

void swap(int *px, int *py)
{
  int temp;

  temp = *px;
  *px = *py;
  *py = temp;
}
```

Parameter mit Referenzen übergeben

Bei der Übergabe mit Zeigern hat, der der die Funktion aufruft Arbeiten zu erledigen (Übergeben der Adresse mit dem Adressoperator) die in die Funktion selbst gehören. Ausserdem führt die mehrfache Dereferenzierung der Zeiger in der Funktion swap eher zu Programmfehlern.
Deshalb bevorzugt man zur Übergabe Referenzen, die sich so einfach verwenden lassen wie normale Variablen und so leistungsfähig wie Zeiger arbeiten:

```
void swap(int &x, int &y);

int main(void)
{
  int x = 5, x = 10;
  swap(x, y);
  return(0);
}

void swap(int &rx, int &ry)
{
  int temp;

  temp = rx;
  rx = ry;
  ry = temp;
}
```

Da bei der Übergabe als Referenz die Originalobjekte abgeändert werden können, lassen sich mehr als eine Information zurückgeben. Diese Lösung umgeht den Rückgabewert der Funktion, der oft für die Meldung von Fehlern vorgesehen wird.

22. Arrays

Ein Array, `myArray[n]`, verfügt über `n` Elemente, die mit `yArray[0]` bis `myArray[n-1]` numeriert sind.

Deklaration

```
long longArray[25];
```

Zugriff

```
longArray[0] = 1;
longArray[1] = 2;
longArray[2] = 3;

...

longArray[24] = 25;
```

Der Arrayname ist ein konstanter Zeiger auf das erste Element des Arrays. Demnach ist `longArray` ein Zeiger auf das Element `&longArray[0]`. Das heisst die erste Adresse des ersten Array - Elements von `longArray`.

Wichtig dabei ist, dass sich ein Array ohne Index wie ein Zeiger verhält und ein Array mit Index wie eine Variable.
So ist es möglich mit `longArray+4` auf die Daten in `longArray[4]` zuzugreifen.

Über das Ende eines Array schreiben

Wenn man in das Element `longArray[50]` schreiben möchte, ignoriert der Compiler die Tatsache, dass dieses Element nicht existiert, berechnet den Offset zum ersten Element und schreibt den Wert über den bereits an dieser Stelle gespeicherten Wert.
An dieser Stelle können beliebige Daten (von anderen Arrays, Variablen, usw.) stehen, so dass das Überschreiben zu unvorhersehbaren Ergebnissen führen kann.

Arrays initialisieren

Bei der Deklaration eines Arrays mit vordefinierten Typen (z.B. Integer) kann man das Array auch initialisieren:

```
int IntegerArray1[5] = {10, 20, 30, 40, 50};
```

Wenn man die Grösse des Arrays nicht angibt, wird das Array entsprechend der Anzahl Initialisierungswerte erzeugt:

```
int IntegerArray2[] = {10, 20, 30, 40, 50};
```
Wir erhalten dasselbe Array wie im ersten Beipsiel.

Initialisiert man mehr Elemente als man hat führt dies zu einem Compilerfehler:

```
int IntegerArray[5] = {10, 20, 30, 40, 50, 60};
```

Man ist nicht gezwungen alle Elemente zu intialisieren. Nicht initialisierte Elemente enthalten keine garantierten Werte, sind aber praktisch immer auf 0 gesetzt.

```
int IntegerArray[5] = {10, 20};
```

Grösse des Arrays ermitteln

Muss man die Grösse eines Arrays ermitteln, kann man den Compiler die Grösse berechnen lassen:

```
const USHORT IntArrayLgth = sizeof(IntArray)/sizeof(IntArray[0]);
```

Teilt man die Gesamtgrösse des Arrays durch die Grösse der einzelnen Elemente erhält man die Anzahl Elemente in einem Array.

Arrays von Objekten

Jedes Objekt lässt sich in einem Array speichern:

Deklaration

```
Car carHouse[50];
```

Zugriff

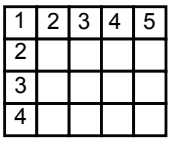
```
carHouse[15].SetYear(1990);
```

23. Mehrdimensionale Arrays

Jede Dimension stellt man durch einen entsprechenden Index im Array dar. Die Anzahl der Dimensionen ist prinzipiell nicht begrenzt. Aus Gründen der Übersicht, arbeitet man fast nie mit mehr als zwei Dimensionen:

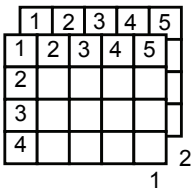
Zweidimensionales Array

```
int IntTwoDim[5][4];
```



Dreidimensionales Array

```
int IntThreeDim[5][4][2];
```



Mehrdimensionale Arrays initialisieren

Um mehrdimensionale Arrays zu initialisieren, weist man den Array - Elementen die Liste der Werte der Reihe nach zu, wobei sich jeweils der letzt Index ändert, während der vorhergehende konstant bleibt:

Beim Array `theArray` kommen z.B. die ersten drei Werte bei `theArray[0]`, die nächsten drei nach `theArray[1]` usw.

```
int theArray[5][3] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}
```

Diese Anweisung lässt sich übersichtlicher darstellen, wenn man die einzelnen Initialisierungen mit geschweiften Klammern gruppiert. Diese inneren geschweiften Klammern ignoriert der Compiler sie dienen nur der Übersichtlichkeit:

```
int theArray[5][3] = { {1,2,3},{4,5,6},
                       {7,8,9},{10,11,12},
                       {13,14,15} };
```

24. Arrays von Zeigern

Arrays speichern ihre Elemente auf dem Stack. Da im Freispeicher, dem Heap, mehr Platz zur Verfügung steht als auf dem Stack kann man jedes Objekt im Heap deklarieren und nur einen Zeiger auf das Objekt im Array ablegen:

Beispiel

```
int main(void)
{
  Car *carHouse[500];          // Array von Zeigern auf den Typ Car
  Car *pCar;                   // Zeiger auf den Typ Car
  int i;

  for(i = 0; i < 500; i++)
  {
    pCar = new Car;            // Neues Car - Objekt auf dem Heap
    pCar->SetYear(1990);
    carHouse[i] = pCat         // Zeiger dem Array zuweisen
  }

  return(0);
}
```

25. Arrays im Heap

Man kann auch das gesamte Array im Freispeicher unterbringen:

```
Car *carHouse = new Car[500];
```

`carHouse` als Zeiger auf das erste Objekt in einem Array von 500 `Car` - Objekten. Anders gesagt zeigt `carHouse` auf das Element `carHouse[0]` bzw. enthält dessen Adresse.

Beispiel

```
Car *carHouse = new Car[500];          // Array im Heap erzeugen
Car *pCar = carHouse;                  // pCar zeigt auf carHouse[0]

pCar->SetYear(1990);                   // setzt carHouse[0] auf 1990
pCar++;                                // geht weiter zu carHouse[1]
pCar->SetYear(1991);                   // setzt carHouse[1] auf 1991

delete [] carHouse;                    // Array im Heap löschen
```

Inkrementiert man den Zeiger, verweist er auf das nächste Objekt. So kann man mit Hilfe der Zeigerarithmetik auf jedes Element von `carHouse` zugreifen.

26. char - Array

Ein String in C++ ist ein Array mit Elementen vom Typ `char` und endet mit einem Null - Zeichen. Die Deklarierung und Initialisierung eines String erfolgt wie bei jedem anderen Array auch:

```
char Greeting[] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Das Nullzeichen '`\0`' wird von vielen Funktionen in C++ als Abschluss-zeichen eines Strings erkannt.

In C++ gibt es eine Kurzform für obige Codezeile:

```
char Greeting[] = "Hello";
```

Deklariert man ein char - Array mit 80 Elementen, kann man darin eine Zeichenfolge mit 79 Zeichen und dem abschliessenden Null - Zeichen aufnehmen.
Liest man Beispielsweise einen String mit `cin` ein und legt ihn in einem char - Array ab, stösst man auf zwei Probleme:

- Hat der eingegebene String mehr Zeichen als das Array Elemente wird über das Ende des Arrays hinausgeschrieben.
- Gibt der Benutzer ein Leerzeichen ein, nimmt `cin` das Ende des Strings an und beendet die Eingabe.

Um diese Probleme zu lösen, gibt es eine spezielle Methode:
`cin.get()`

Beispiel

```
int main()
{
  char buffer[80];
  cin.get(buffer, 79);        // Bis maximal 79 Zeichen oder bis
                              // newline einlesen.
  return(0);
}
```

27. strcpy() und strncpy()

C++ erbt von C eine Funktionsbibliothek `(string.h)` für die Behandlung von Strings, darunter auch die Funktionen `strcpy()` und `strncpy()`.

Die Funktion `strcpy()` kopiert den gesamten Inhalt eines Strings in den angegebenen Puffer:

```
#include <string.h>

int main()
{
  char String1[] = "Keiner lebt fuer sich allein.";
  char String2[80];

  strcpy(String1, String2);

  return(0);
}
```

Ist die Quelle `(String1)` grösser als das Ziel `(String2)` schreibt `strcpy` über das Ende des Puffers hinaus.

Mit der Funktion `strncpy` kann man sich dagegen schützen, da sie auch den Maximalwert der zu kopierenden Zeichen übernimmt:

```
...
const int MaxLength = 80;
char String1[] = "Keiner lebt fuer sich allein.";
char String2[MaxLength + 1];

strncpy(String1, String2, MaxLength);
...
```

Da die Funktionen `strcpy()` und `strncpy()` automatisch ein Null - Zeichen an das Ende des Strings anfügen, muss der Puffer `String2` `MaxLength + 1` Zeichen enthalten.

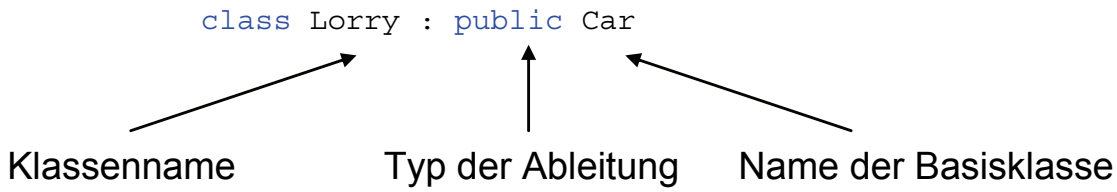
28. Vererbung

Leitet man eine neue Klasse `Lorry` (Lastwagen) von der Klasse `Car` (Fahrzeug) ab, muss man nicht explizit feststellen, dass Lastwagen fahren und stoppen, da sie diese Eigenschaften von `Car` erben. Da eine `Lorry` - Klasse von einer `Car` - Klasse erbt, fährt und stoppt `Lorry` automatisch.

Eine Klasse, die eine existierende Klasse um neue Funktionalität erweitert, bezeichnet man als von dieser Originalklasse abgeleitet. Die Originalklasse heisst Basisklasse der neuen Klasse.

Syntax der Ableitung

Die Basisklasse muss vor der Ableitung deklariert werden, da sonst der Compiler Fehler erhält.



Beispiel

```
class Car                       // Basisklasse Car
{
  public:
    void Car();                 // Konstruktoren
    void Car(int y);
    void ~Car();

    void Drive();               // Andere Methoden
    void Stop();

    void SetYear(int y);        // Zugriffsfunktionen
    int GetYear();
    int GetMaxVelocity()

  protected:
    int year
};

class Lorry : public Car        // Von Car abgeleitete Klasse Lorry
{
  public:
    void Lorry();               // Konstruktoren
    void ~Lorry();
    void LoadThings();          // Andere Methoden
};
```

`Lorry` - Objekte erben die Variablen `(year)` sowie alle Methoden von der Klasse Car. Ausgenommen hiervon sind der Kopierkonstruktor, die Konstruktoren und der Destruktor.

Protected

Klassendaten, die als `private` deklariert sind, sind für abgeleitet Klassen nicht verfügbar. Da man die Daten aber auch nicht als `public` deklarieren will, da sonst auch andere Klassen, nicht nur die abgeleiteten, Zugriff hätten, gibt es das Zugriffsschlüsselwort `protected`.

Als `protected` deklarierte Datenelemente und Funktionen sind für abgeleitete Klassen vollständig sichtbar, sonst aber privat.

Konstruktoren und Destruktoren

Beim erzeugen vom Objekt `Scania` wird zuerst dessen Konstruktor aufgerufen, der ein `Car` - Objekt erzeugt. Dann folgt der Aufruf des `Lorry` - Konstruktors, der die Konstruktion des `Lorry` - Objekts vervollständigt.
`Scania` existiert erst, nachdem dieses Objekt vollständig erzeugt wurde, das heisst, sowohl der `Car` - Teil als auch der `Lorry` - Teil aufgebaut ist. Daher ist der Aufruf beider Konstruktoren erforderlich.

Beim zerstören des `Scania` - Objekts wird zuerst der `Lorry` - Destruktor und dann der Destruktor für den `Car` - Teil von `Scania` aufgerufen.

Funktionen redefinieren

Ein `Lorry` - Objekt hat auf alle Elementfunktionen in der Klasse Car sowie auf alle Elementfunktionen, die die Deklaration in der `Lorry` - Klasse gegebenenfalls hinzufügt `(LoadThings)`.

Ein `Lorry` - Objekt kann auch eine Funktion der Basisklasse redefinieren, das heisst, die Implementierung einer Funktion der Basisklasse in einer abgeleiteten Klasse ändern.
Wird ein Objekt der abgeleiteten Klasse erstellt, wird die korrekte Funktion aufgerufen.

Man spricht von Redefinieren, wenn eine abgeleitete Klasse eine Funktion mit demselben Rückgabewert und derselben Signatur wie eine Elementfunktion in der Basisklasse, aber mit einer neuen Implementierung, erzeugt:

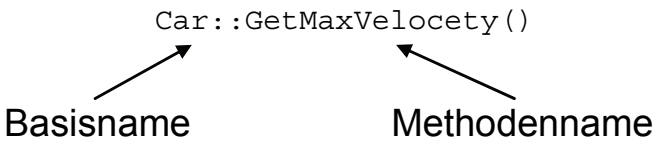
```
class Car                                    // Basisklasse Car
{
  public:
    ...
    int GetMaxVelocity() {return(200);}
    ...
};

class Lorry : public Car                     // Ableitung von Car
{
  public:
    ...
    int GetMaxVelocity() {return(120);}      // Redefinition
    ...
};
```

Methoden der Basisklasse verbergen

Im obigen Beispiel verbirgt die Methode `GetMaxVelocity` der Klasse `Lorry` die Methode der Basisklasse `Car`.
Es ist zu beachten, dass wenn eine Methode der Basisklasse redefiniert wird auch nicht mehr ohne weiteres auf die überladenen Funktionen der redefinierten Funktion zugegriffen werden kann.

Basismethoden aufrufen

Eine redefinierte Basismethode kann weiterhin aufgerufen werden, wenn man den vollständigen Namen der Methode angibt:



Also ruft man so die redefinierte Methode von `Lorry` auf:

```
Lorry.GetMaxVelocity();
```

und so die Basismethode der Basisklasse:

```
Lorry.Car::GetMaxVelocity();
```

29. Virtuelle Methoden:

In C++ lassen sich abgeleitete Klassenobjekte an Zeiger auf die Basisklassen zuweisen (erweiterte Polymorphie):

```
Car* pLorry = new Lorry;
```

Diese Anweisung erzeugt ein neues `Lorry` - Objekt auf dem Heap. Der von `new` zurückgegebene Zeiger auf dieses Objekt wird einem Zeiger auf `Car` zugewiesen.
Das ist durchaus sinnvoll, da ein Lastwagen `(Lorry)` ein Fahrzeug `(Car)` ist.

Über diesen Zeiger kann man jede Methode in `Car` aufrufen. Natürlich möchte man, dass die in `Lorry` redefinierten Methoden die korrekte Funktion aufrufen.
Virtuelle Elementfunktionen erlauben genau das.

Beispiel

```
// Einsatz virtueller Methoden.

#include <iostream.h>

class Car
{
  public:
    Car() {cout << "Car-Konstruktor...\n";}
    ~Car() {cout << "Car-Destruktor...\n";}

    // Virtuelle Methode.

    virtual void GetMaxVelocety() const {cout << "200 km/h\n";}
};

class Lorry: public Car
{
  public:
    Lorry() {cout << "Lorry-Konstruktor...\n";}
    ~Lorry () {cout << "Lorry-Destruktor...\n";}

    // Redefinition der virtuellen Methode.

    void GetMaxVelocety() const { cout << "120 km/h\n";}
};

int main(void)
{
  Car *pLorry = new Lorry;
```

```
  pLorry-> GetMaxVelocety ();

  delete pLorry;
  return(0);
}
```

Ausgabe

```
Car-Konstruktor...
Lorry-Konstruktor...
120 km/h
Car-Destruktor...
```

Werden von der Klasse `Lorry` noch andere Objekte abgeleitet, sollte die Elementmethode `GetMaxVelocety()` ebenfalls virtuell sein, damit sie in abgeleiteten Objekten auch redefiniert werden kann.

Virtuelle Destruktoren

Beim obigen Beispiel haben wir gesehen, dass beim Zerstören des Objekts nur der `Car` - Destruktor aufgerufen wurde. Von `Lorry` reservierter Speicher würde also nicht freigegeben.
Verwendet man einen virtuellen Destruktor, sobald eine Methode virtuell ist, kann dies nicht geschehen:

```
// Einsatz von einem virtuellen Destruktor.

#include <iostream.h>

class Car
{
  public:
    Car() {cout << "Car-Konstruktor...\n";}

    // Virtueller Destruktor

    virtual ~Car() {cout << "Car-Destruktor...\n";}
    ...
};

class Lorry: public Car
{
  public:
    Lorry() {cout << "Lorry-Konstruktor...\n";}
    ~Lorry () {cout << "Lorry-Destruktor...\n";}
    ...
};
...
```

Ausgabe

```
Car-Konstruktor...
Lorry-Konstruktor...
120 km/h
Lorry-Destruktor...
Car-Destruktor...
```

Werden von der Klasse `Lorry` noch andere Objekte abgeleitet, sollte ebenfalls der Destruktor von Lorry `(~Lorry())` virtuell sein, damit der Destruktor von den abgeleiteten Objekten auch ausgeführt wird.

Datenteilung (Slicing)

Virtuelle Funktionen funktionieren nur bei Zeigern und Referenzen. Die Übergabe eines Objekts als Wert lässt den Aufruf virtueller Elementfunktionen nicht zu.

Beispiel

```
// Einsatz virtueller Methoden.

#include <iostream.h>

class Car
{
  public:
    Car() {cout << "Car-Konstruktor...\n";}
    virtual ~Car() {cout << "Car-Destruktor...\n";}

    // Virtuelle Methode.
    virtual void GetMaxVelocety() const {cout << "200 km/h\n";}
};

class Lorry: public Car
{
  public:
    Lorry() {cout << "Lorry-Konstruktor...\n";}
    ~Lorry () {cout << "Lorry-Destruktor...\n";}

    // Redefinition der virtuellen Methode.
    void GetMaxVelocety() const { cout << "120 km/h\n";}
};

void ValueFunction(Car CarValue);
void PtrFunction(Car *pCar);
void RefFunction(Car &rCar)
```

```
int main(void)
{
  Car *pLorry = new Lorry;
  ValueFunction(*pLorry);     // Übergabe als Wert
  PtrFunction(pLorry);        // Übergabe als Zeiger
  RefFunction(*pLorry);       // Übergabe als Referenz

  delete pLorry;
  return(0);
}

void ValueFunction(Car CarValue)
{
  CarValue. GetMaxVelocety ();
}
void PtrFunction(Car *pCar)
{
  pCar-> GetMaxVelocety ();
}

void RefFunction(Car &rCar)
{
  rCar. GetMaxVelocety ();
}
```

Ausgabe

```
Car-Konstruktor...
Lorry-Konstruktor...
200 km/h
120 km/h
120 km/h
Lorry-Destruktor...
Car-Destruktor...
```

In der ersten Funktion `(ValueFunction)` wird das Objekt `Lorry` als Wert übergeben. Deshalb kann nicht auf die virtuelle Funktion `(GetMaxVelocety())` von `Lorry` zugegriffen werden und die Ausgabe lautet `200 km/h`.
Bei den beiden anderen Funktionen `(PtrFunction, RefFunction)` wird das Objekt als Zeiger oder als Referenz übergeben.
Somit ist auch der Aufruf virtueller Elementfunktionen zulässig (→ `120 km/h`).

30. Dynamische Typenumwandlung

Wenn man eine Elementmethode in eine abgeleitete Klasse hinzufügt, die für die Basisklasse nicht passend ist und deshalb nicht virtuell imp-

lementiert wird, kann man diese nicht über den Zeiger vom Typ der Basisklasse aufrufen.

Um dennoch auf diese Elementmethode zugreifen zu können, muss man eine dynamische Typenumwandlung vornehmen. Man wandelt also den Zeiger der Basisklasse in den abgeleiteten Typ um.
Dazu benutzt man den Operator `dynamic_cast`.

Beispiel

```
// Dynamische Typenumwandlung

#include <iostream.h>

class Car
{
  public:
    Car() {cout << "Car-Konstruktor...\n";}
    virtual ~Car() {cout << "Car-Destruktor...\n";}

    virtual void GetMaxVelocety() const {cout << "200 km/h\n";}
};

class Lorry: public Car
{
  public:
    Lorry() {cout << "Lorry-Konstruktor...\n";}
    ~Lorry() {cout << "Lorry-Destruktor...\n";}

    void GetMaxVelocety() const {cout << "120 km/h\n";}
    void LoadThings() const {cout << "Waren einladen...\n";}
};

int main(void)
{
  Car *pCar = new Lorry;
  Lorry *pRealLorry = 0;

  pCar->GetMaxVelocety();

  // Dynamische Typenumwandlung
  pRealLorry = dynamic_cast <Lorry *> (pCar);

  if(pRealLorry != 0)
    pRealLorry->LoadThings();

  delete pRealLorry;
  delete pCar;

  return(0);
}
```

Ausgabe

```
Car-Konstruktor...
Lorry-Konstruktor...
120 km/h
Waren einladen...
Lorry-Destruktor...
Car-Destruktor...
```

Um die Elementmethode `LoadThings()` von der abgeleiteten Klasse `Lorry` aufrufen zu können, wird ein `Lorry` - Zeiger erzeugt und die Umwandlung mit dem Operator `dynamic_cast` vorgenommen.

Ist die Umwandlung erfolgreich erhält man einen neuen einwandfreien Lorry - Zeiger. Wenn überhaupt kein Lorry - Objekt existiert und die Umwandlung scheitert, bekommt man einen Null - Zeiger zurück. Deshalb sollte man den erhaltenen Zeiger immer auf Null testen, bevor man ihn weiterverwendet.

31. Abstrakte Datentypen

Abstrakte oder "rein virtuelle" Funktionen lassen sich in C++ erzeugen, indem die Funktion als `virtual` deklariert und mit 0 initialisiert wird:

```
virtual void GetMaxVelocety() = 0;
```

Jede Klasse mit einer oder mehreren abstrakten Funktionen ist ein abstrakter Datentyp (ADT). Es ist nicht möglich, ein Objekt von einer als ADT fungierenden Klasse zu instantiieren.

Da von ADTs abgeleitete Klassen die abstrakten Funktionen erben, muss man diese erst redefinieren, wenn man Objekte dieser Klasse instantiieren will.
Man muss also auf jeden Fall die abstrakte Funktion einmal redefinieren.

Beispiel

```
// Abstrakte Funktionen

#include <iostream.h>
// abstrakter Datentyp
class Car
```

```
{
  public:
    Car() {cout << "Car-Konstruktor...\n";}
    virtual ~Car() {cout << "Car-Destruktor...\n";}

    // Abstrakte Methode.
    virtual void GetMaxVelocety() const = 0;
};

class Lorry: public Car
{
  public:
    Lorry() {cout << "Lorry-Konstruktor...\n";}
    ~Lorry () {cout << "Lorry-Destruktor...\n";}

    // Redefinition der virtuellen Methode.

    void GetMaxVelocety() const { cout << "120 km/h\n";}
};

int main(void)
{
  Car *pLorry = new Lorry;
  pLorry->GetMaxVelocety();

  delete pLorry;
  return(0);
}
```

Ausgabe

```
Car-Konstruktor...
Lorry-Konstruktor...
120 km/h
Lorry-Destruktor...
Car-Destruktor...
```

Da niemals Objekte eines ADTs erzeugt werden, gibt es auch keinen Grund Implementierungen zur Verfügung zu stellen. ADTs dienen also nur als Schnittstelle auf Objekte, die von ihnen abgeleitet sind.

Es ist aber möglich, eine Implementierung für eine abstrakte Funktion anzugeben. Die Funktion lässt sich dann von den abgeleiteten Objekten aufrufen, um z.B. allen redefinierten Funktionen eine gemeinsame Funktionalität zu verleihen:

```
class Car
{
  public:
    ...
    // Abstrakte Methode.
```

```
    virtual void GetMaxVelocety() const = 0;
};

// Implementierung für eine abstrakte Methode
void Car::GetMaxVelocety() const
{
  cout << "200 km/h";
}
class Lorry: public Car
{
  public:
    ...
    // Redefinition der virtuellen Methode.
    void GetMaxVelocety() const { cout << "120 km/h\n";}
};

void Lorry::GetMaxVelocety() const
{
  // Aufruf der abstrakten Methode von Car
  Car::GetMaxVelocety();
}
...
```

32. Statische Datenelemente

Will man Daten verfolgen, die mehrere Objekte einer Klasse gemeinsam nutzen, z.B. wie viele Car - Objekte bisher erzeugt wurden und wie viele es noch gibt, benötigt man statische Datenelemente.

Im Gegensatz zu normalen Elementvariablen werden statische Elementvariablen von allen Instanzen einer Klasse gemeinsam genutzt.

Beispiel

```
// Statische Datenelemente

#include <iostream.h>

class Car
{
  public:
    Car(){HowManyCars ++;
          cout << "Car-Konstrukotr...\n";}
    ~Car(){HowManyCars --;
           cout << "Car-Destruktor...\n";}

    void SetYear(int y){year = y;}
    int GetYear(){return(year);}
    static int HowManyCars;
```

```
  private:
    int year;
};

int Car::HowManyCars = 0;

int main(void)
{
  Car *CarHouse[5];  int i;

  for(i = 0; i < 5; i++)
    CarHouse[i] = new Car;

  cout << "Es hat " << Car::HowManyCars << " Wagen.\n";

  delete CarHouse[0];

  cout << "Jetzt hat es " << Car::HowManyCars << " Wagen.\n";

  delete CarHouse[1];

  cout << "Jetzt hat es " << Car::HowManyCars << " Wagen.\n";

  delete CarHouse[2];
  delete CarHouse[3];
  delete CarHouse[4];

  cout << "Jetzt hat es " << Car::HowManyCars << " Wagen.\n";

  return(0);
}
```

Ausgabe

```
Car-Konstrukotr...
Car-Konstrukotr...
Car-Konstrukotr...
Car-Konstrukotr...
Car-Konstrukotr...
Es hat 5 Wagen.
Car-Destruktor...
Jetzt hat es 4 Wagen.
Car-Destruktor...
Jetzt hat es 3 Wagen.
Car-Destruktor...
Car-Destruktor...
Car-Destruktor...
Jetzt hat es 0 Wagen.
```

Unschön ist hierbei, dass `HowManyCars` öffentlich ist und `main` direkt darauf zugreift. Es ist besser die statische Elementvariable mit den anderen Variablen als privat zu deklarieren und eine öffentliche Zu-

griffsmethode bereitzustellen, solange man auf die Daten über eine Instanz von `Car` zugreift.
Will man andererseits auf diese Daten direkt zugreifen, ohne dass unbedingt ein Car - Objekt verfügbar sein muss, gibt es zwei Möglichkeiten: die Variable öffentlich halten, oder eine statische Elementfunktion (siehe Kap. 32) bereitzustellen.

33. Statische Elementfunktionen

Statische Elementfunktionen sind vergleichbar mit statischen Elementvariablen. Sie existieren nur in einem Objekt, aber im Gültigkeitsbereich der Klasse. Demzufolge kann man sie aufrufen, ohne ein Objekt dieser Klasse verfügbar zu haben.

Beispiel

```
// Statische Elementfunktionen

#include <iostream.h>

class Car
{
  public:
    Car(){HowManyCars ++;
          cout << "Car-Konstrukotr...\n";}
    ~Car(){HowManyCars --;
           cout << "Car-Destruktor...\n";}

    void SetYear(int y){year = y;}
    int GetYear(){return(year);}
    static int GetHowMany(){cout<<"Statische Elementfunktion.\n";
                            return(HowManyCars);}

  private:
    int year;
    static int HowManyCars;
};

int Car::HowManyCars = 0;

int main(void)
{
  Car *CarHouse[5];  int i;

  for(i = 0; i < 5; i++)
    CarHouse[i] = new Car;

  cout << "Es hat " << Car::GetHowMany() << " Wagen.\n";
```

```
  delete CarHouse[0];

  cout << "Jetzt hat es " << Car::GetHowMany() << " Wagen.\n";

  delete CarHouse[1];

  cout << "Jetzt hat es " << Car::GetHowMany() << " Wagen.\n";

  delete CarHouse[2];
  delete CarHouse[3];
  delete CarHouse[4];

  cout << "Jetzt hat es " << Car::GetHowMany() << " Wagen.\n";

  return(0);
}
```

Ausgabe

```
Car-Konstrukotr...
Car-Konstrukotr...
Car-Konstrukotr...
Car-Konstrukotr...
Car-Konstrukotr...
Statische Elementfunktion.
Es hat 5 Wagen.
Car-Destruktor...
Statische Elementfunktion.
Jetzt hat es 4 Wagen.
Car-Destruktor...
Statische Elementfunktion.
Jetzt hat es 3 Wagen.
Car-Destruktor...
Car-Destruktor...
Car-Destruktor...
Statische Elementfunktion.
Jetzt hat es 0 Wagen.
```

Da die Funktion `GetHowMany()` öffentlich ist, kann man darauf mit jeder Funktion zugreifen. Durch ihren statischen Charakter benötigt man kein Objekt vom Typ `Car`, auf dem man diese Funktion aufruft.

Natürlich könnte man `GetHowMany()` auch auf den in `main()` verfügbaren Car - Objekten aufrufen, genau wie bei allen anderen Zugriffsfunktionen.

Da statische Elementfunktionen keinen `this` - Zeiger haben, können sie nicht als const deklariert werden. Da ausserdem der Zugriff auf Datenelemente in Elementfunktionen mit dem Zeiger `this` erfolgt,

können statische Methoden, nicht auf nicht statische Elementvariablen zugreifen.

34. Zeiger auf Funktionen

Genau wie ein Array - Name ein konstanter Zeiger auf das Erste Element des Arrays ist, stellt der Funktionsname einen konstanten Zeiger auf die Funktion dar. Man kann also eine Variable als Zeiger auf eine Funktion deklarieren und die Funktion mit Hilfe dieses Zeigers aufrufen.

Deklaration

```
long * Function (int);
long (* funcPtr) (int);
```

Die erste Deklaration, `Function`, ist eine Funktion, die einen Integer übernimmt und einen Zeiger auf eine Variable von Typ `long` zurückgibt. Die zweite Deklaration, `funcPtr`, ist ein Zeiger auf eine Funktion, die einen Integer übernimmt und eine Variable von Typ `long` zurückgibt.

Beispiel

```
// Zeiger auf Funktionen

#include <iostream.h>

void Cube (int &, int &);
void Swap (int &, int &);

int main(void)
{
  void (*pFunc) (int &, int &);
  int value1, value2;

  cout << "Wert 1 eingeben: ";
  cin >> value1;
  cout << "Wert 2 eingeben: ";
  cin >> value2;

  pFunc = Cube;
  pFunc(value1, value2);

  cout << "\nWert1 = " << value1 << "  Wert2 = " << value2;
```

```
  pFunc = Swap;
  pFunc(value1, value2);

  cout << "\nWert1 = " << value1 << "  Wert2 = " << value2;

  return(0);
}

void Cube (int & rX, int & rY)
{
  int temp;
  temp = rX;
  rX = rX * rX;
  rX = rX * temp;

  temp = rY;
  rY = rY * rY;
  rY = rY * temp;
}

void Swap (int & rX, int & rY)
{
  int temp;

  temp = rX;
  rX = rY;
  rY = temp;
}
```

Ausgabe

```
Wert 1 eingeben: 2
Wert 2 eingeben: 4

Wert1 = 8  Wert2 = 64
Wert1 = 64  Wert2 = 8
```

Kurzaufruf

Den Zeiger auf eine Funktion muss man nicht dereferenzieren, obwohl man es durchaus tun kann. Angenommen, pFunc ist ein Zeiger auf eine Funktion, die einen Integer übernimmt und eine Variable von Typ long zurückgibt, und man weist pFunc eine passende Funktion zu, kann man diese Funktion entweder mit

```
pFunc(x);
```

oder mit

```
(*pFunc)(x);
```

aufrufen. Beide Formen sind identisch. Die erste ist eine Kurzversion der zweiten.

Arrays mit Zeigern auf Funktionen

Genau wie man ein Array mit Zeigern auf Integer deklarieren kann, lässt sich auch ein Array mit Zeigern auf Funktionen deklarieren, die einen bestimmten Wertetyp zurückgeben.

Beispiel

```
// Array von Zeigern auf Funktionen

#include <iostream.h>

void Cube (int &, int &);
void Swap (int &, int &);

int main(void)
{
  void (*pFuncArray[2]) (int &, int &);
  int value1, value2;

  cout << "Wert 1 eingeben: ";
  cin >> value1;
  cout << "Wert 2 eingeben: ";
  cin >> value2;

  pFuncArray[0] = Cube;
  pFuncArray[1] = Swap;

  pFuncArray[0](value1, value2);

  cout << "\nWert1 = " << value1 << "  Wert2 = " << value2;

  pFuncArray[1](value1, value2);

  cout << "\nWert1 = " << value1 << "  Wert2 = " << value2;

  return(0);
}

void Cube (int & rX, int & rY)
{
  int temp;
  temp = rX;
  rX = rX * rX;
  rX = rX * temp;

  temp = rY;
```

```
  rY = rY * rY;
  rY = rY * temp;
}

void Swap (int & rX, int & rY)
{
  int temp;

  temp = rX;
  rX = rY;
  rY = temp;
}
```

Ausgabe

```
Wert 1 eingeben: 2
Wert 2 eingeben: 4

Wert1 = 8  Wert2 = 64
Wert1 = 64  Wert2 = 8
```

Zeiger auf Funktionen an andere Funktionen übergeben

Die Zeiger auf Funktionen (inklusive Arrays mit Zeigern auf Funktionen) kann man an andere Funktionen übergeben, die Aktionen ausführen und dann die richtige Funktion unter Verwendung des Zeigers ausführen.

Beispiel

```
// Übergabe von Zeigern auf Funktionen als Funktionsargumente

#include <iostream.h>

void Cube (int &, int &);
void Swap (int &, int &);
void PrintValues(void (*)(int &, int &), int &, int &);

int main(void)
{
  void (*pFunc) (int &, int &);
  int value1, value2;

  cout << "Wert 1 eingeben: ";
  cin >> value1;
  cout << "Wert 2 eingeben: ";
  cin >> value2;

  pFunc = Cube;
```

```
  PrintValues(pFunc, value1, value2);

  pFunc = Swap;

  PrintValues(pFunc, value1, value2);


  return(0);
}
void PrintValues(void (*pFunc)(int &, int &), int &x, int &y)
{
  pFunc(x, y);
  cout << "\nWert1 = " << x << "  Wert2 = " << y;
}

void Cube (int & rX, int & rY)
{
  int temp;
  temp = rX;
  rX = rX * rX;
  rX = rX * temp;

  temp = rY;
  rY = rY * rY;
  rY = rY * temp;
}

void Swap (int & rX, int & rY)
{
  int temp;

  temp = rX;
  rX = rY;
  rY = temp;
}
```

Ausgabe

```
Wert 1 eingeben: 2
Wert 2 eingeben: 4

Wert1 = 8  Wert2 = 64
Wert1 = 64  Wert2 = 8
```

typedef bei Zeigern auf Funktionen

Die Konstruktion `void (*)(int &, int &)` ist unhandlich. Das ganze lässt sich mit `typedef` vereinfachen. Man deklariert einen Typ (z.B. `VPF`) als Zeiger auf eine Funktion, die `void` zurückgibt und zwei Integer-Referenzen übernimmt.

Hier das Beispiel von oben mit typedef neu geschrieben:

```
// Einsatz von typedef

#include <iostream.h>

void Cube (int &, int &);
void Swap (int &, int &);
void PrintValues(void (*)(int &, int &), int &, int &);
typedef void (*VPF)(int &, int &);

int main(void)
{
  VPF pFunc;
  int value1, value2;

  cout << "Wert 1 eingeben: ";
  cin >> value1;
  cout << "Wert 2 eingeben: ";
  cin >> value2;

  pFunc = Cube;

  PrintValues(pFunc, value1, value2);

  pFunc = Swap;

  PrintValues(pFunc, value1, value2);


  return(0);
}

void PrintValues(VPF pFunc, int &x, int &y)
{
  pFunc(x, y);
  cout << "\nWert1 = " << x << "  Wert2 = " << y;
}

void Cube (int & rX, int & rY)
{
  int temp;
  temp = rX;
  rX = rX * rX;
  rX = rX * temp;

  temp = rY;
  rY = rY * rY;
  rY = rY * temp;
}

void Swap (int & rX, int & rY)
{
  int temp;

  temp = rX;
```

```
  rX = rY;
  rY = temp;
}
```

Ausgabe

```
Wert 1 eingeben: 2
Wert 2 eingeben: 4

Wert1 = 8  Wert2 = 64
Wert1 = 64  Wert2 = 8
```

35. Zeiger auf Elementfunktionen

Bei einem Zeiger auf eine Elementfunktion verwendet man die gleiche Syntax wie bei einem Zeiger auf eine Funktion, schliesst aber den Klassennamen und den Zugriffsoperator (::) ein:

```
void (Car::*pFunc)(int, int);
```

Beispiel

```
// Zeiger auf Elementfunktionen

#include <iostream.h>

class Car
{
  public:
    void Drive(){cout << "Fahren...\n";}
    void Stop(){cout << "Stoppen...\n";}
};

int main(void)
{
  void (Car::*pFunc)();
  Car *ptr = new Car;

  pFunc = Car::Drive;
  (ptr->*pFunc)();

  pFunc = Car::Stop;
  (ptr->*pFunc)();

  delete ptr;
  return(0);
}
```

Ausgabe

```
Fahren...
Stoppen...
```

Arrays von Zeigern auf Elementfunktionen

Genau wie bei Zeigern auf Funktionen lassen sich Zeiger auf Elementfunktionen in einem Array speichern.
Das Array kann mit den Adressen der verschiedenen Elementfunktionen initialisiert werden, und diese lassen sich durch Indizierung des Arrays aufrufen.

Beispiel

```
// Array mit Zeigern auf Elementfunktionen

#include <iostream.h>

class Car
{
  public:
    void Drive(){cout << "Fahren...\n";}
    void Stop(){cout << "Stoppen...\n";}
};

typedef void (Car::*PDF)();

int main(void)
{
  PDF CarFunctions[2] = {Car::Drive, Car::Stop};
  Car *ptr = new Car;

  (ptr->*CarFunctions[0])();

  (ptr->*CarFunctions[1])();

  delete ptr;
  return(0);
}
```

Ausgabe

```
Fahren...
Stoppen...
```

36. Präprozessor

Bei jeder Ausführung des Compilers startet als erstes der Präprozessor. Dieser sucht nach Anweisungen, die mit einem Nummerzeichen (#) beginnen. Derartige Anweisungen bewirken eine Änderung des Quellcodes. Im Ergebnis entsteht eine neue Quellcodedatei (Zwischendatei), die anschliessend der Compiler liest und compiliert.

#include

Die Anweisung `#include` teilt dem Präprozessor mit, die Datei mit dem auf die `#include` - Direktive folgenden Namen zu suchen und sie an dieser Stelle in die Zwischendatei zu schreiben. Das verhält sich genauso, wie wenn man selbst die gesamte Datei direkt in die Quellcodedatei geschrieben hätte.

```
#include <stdio.h>
```

#define

Der Befehl `#define` definiert eine String-Erestzung. Schreibt man zum Beispiel

```
#define BIG 512
```

hat man den Präprozessor angewiesen, den String `BIG` an allen Stellen im Quelltext gegen den String `512` auszutauschen.

Schreibt man:

```
#define BIG 512
int myArray[BIG];
```

sieht die von Präprozessor produzierte Zwischendatei so aus:

```
int myArray[512];
```

Konstanten mit #define

siehe Kapitel 4.

#undef

Wenn man Namen definiert hat und diese Definition aus dem Code heraus aufheben möchte, kann man die Anweisung `#undef` verwenden, die entgegengesetzt zur Anweisung `#define` arbeitet.

#ifdef, #ifndef, #else, #endif

Die Präprozessor-Anweisungen `#ifdef` und `#ifndef` testen, ob ein String definiert wurde bzw. nicht definiert wurde. Bei beiden Befehlen muss vor dem Blockende ein abschliessender `#endif`-Befehl erscheinen.

```
#define DEBUG

#ifdef DEBUG
  cout << "DEBUG definiert";
#else
  cout << "DEBUG nicht definiert";
#endif
```

Trifft der Präprozessor auf `#ifdef`, durchsucht er eine von ihm angelegte Tabelle, ob die Zeichenfolge `DEBUG` definiert ist. Ist dies der Fall, liefert `#ifdef` das Ergebnis `True`, und alles bis zum nächsten `#else` oder `#endif` wird in die Zwischendatei zur Kompilierung geschrieben.

Das logische Gegenstück von `#ifdef` ist `#ifndef`. Diese Anweisung liefert `True`, wenn die Zeichenfolge bis zu diesem Punkt noch nicht in der Datei definiert wurde.

Fehlersuche (Debugging)

Es ist üblich, speziellen Code zur Fehlersuche zwischen `#ifdef DEBUG` und `#endif` einzuschliessen. Damit lässt sich der gesamte Debugging-Code leicht aus der Quelldatei entfernen, wenn man die endgültige Version kompiliert: man definiert einfach nicht den Begriff `DEBUG`.

Schutz vor Mehrfachdeklaration

Annahme: Die Klasse Car ist in der Datei Car.h deklariert. Die von Car abgeleitete Klasse Bus muss die Datei Car.h in Bus.h einbinden, da sich Bus sonst nicht von Car ableiten lässt. Der Lorry-Header schliesst Car.h aus dem selben Grund ein.
Wenn man nun eine Methode erzeugt, die sowohl Lorry als auch Bus verwendet, läuft man Gefahr, Car.h zweimal einzubinden, was zu einem Compiler-Fehler führt.
Dieses Problem lässt sich mit einer Schutzvorkehrung lösen:

```
#ifndef Car_h
#define Car_h
...             // Hier steht der gesamte Inhalt der Datei
#endif
```

Wenn das Programm erstmalig diese Datei einbindet und der Compiler die erste Zeile auswertet, liefert der Test `True`. Das heisst, `Car_h` ist noch nicht definiert. Demzufolge holt der Compiler jetzt die Definition von `Car_h` nach und bindet die gesamte Datei ein.
Will der Compiler die Datei ein zweites Mal einbinden liefert der Test `False`, da `Car_h` nun definiert ist. So springt das Programm zum nächsten `#else` oder, wenn dieses fehlt, zum nächsten `#endif` (hier: Ende der Datei).

Der Name des zu definierten Symbols spielt keine Rolle.
Üblich ist:

```
DATEINAME_DATEIENDUNG
```

37. Makros

Ein Makro ist ein Symbol, das mit Hilfe von #define erzeugt wurde und in der Art einer Funktion ein Integer-Argument übernimmt.
Der Präprozessor substituiert den Ersetzungsstring durch das jeweils übergebene Argument.

Ein Makro wird wie folgt definiert:

```
#define TWICE(x) ((x) * 2)
```

Im Code schreibt man dann

```
TWICE(4)
```

Der gesamte String `TWICE(4)` wird entfernt und durch den Wert `8` ersetzt. Trifft der Präprozessor auf die `4`, setzt er dafür `((4) * 2)`, was dann zu `4 * 2` oder `8` ausgewertet wird.

Ein Makro kann über mehrere Parameter verfügen:

```
#define MAX(x,y) ((x) > (y) ? (x) : (y))
#define MIN(x,y) ((x) < (y) ? (x) : (y))
```

`MAX` und `MIN` verwenden:

```
int x = 5, y = 7, z, o;
z = MAX(x,y);
o = MIN(x,y);
```

Nach der Ausführung des Präprozessor (Zwischendatei):

```
int x = 5, y = 7, z, o;
z = 7;
o = 5;
```

Es ist zu beachten, dass in einer Makro-Definition die öffnende Klammer unmittelbar nach dem Makronamen folgen muss (ohne Lehrzeichen dazwischen). Sonst nimmt der Präprozessor eine einfache Textersetzung vor.
`MAX` wird durch `(x,y) ((x) > (y) ? (x) : (y))` ersetzt, woran sich das nach `MAX` angegeben `(x,y)` anschliesst.

Der Code der Zwischendatei würde dann wie folgt aussehen:

```
int x = 5, y = 7, z, o;
z = (x,y) ((x) > (y) ? (x) : (y)) (x,y);
o = (x,y) ((x) < (y) ? (x) : (y)) (x,y);
```

Der Präprozessor ist nicht darauf angewiesen, dass Klammern um die Argumente in der Ersetzungszeichenfolge stehen. Allerdings helfen die Klammern unerwünschte Nebeneffekte zu vermeiden:

```
#define CUBE1(a) ((a) * (a) * (a))
#define CUBE2(a) a * a * a
```

Übergibt man den Wert `5` liefern beide Makros `125` als Ergebnis.
Wird aber `5 + 7` als Parameter übergeben, liefert `CUBE1(5 + 7)` die Erweiterung

```
((5 + 7) * (5 + 7) * (5 + 7))
```

damit

```
((12) * (12) * (12))
```

und führt zum Ergebnis `1728`.

Dagegen wird `CUBE2(5 + 7)` zu

```
5 + 7 * 5 + 7 * 5 + 7
```

erweitert. Da die Multiplikation einen höheren Vorrang als die Addition hat ergibt sich

```
5 + (7 * 5) + (7 * 5) + 7
```

damit

```
5 + (35) + (35) + 7
```

was schliesslich das Ergebnis 82 liefert.

String-Manipulation

Der Operator zur Zeichenkettenbildung (`#`, stringizing operator) schliesst alle Zeichen, die bis zum nächsten Whitespace auf den Operator folgen, in Anführungszeichen ein:

```
#define WRITESTRING(x) cout << #x
```

Aufruf:

```
WRITESTRING(Das ist ein String);
```

Der Präprozessor wandelt dann den Code wie folgt um:

```
cout << "Das ist ein String";
```

Mit dem Verkettungsoperator lassen sich mehrere Terme zu einem neuen Wort verbinden:

```
#define fPRINT(x) f ## x ## Print
```

`fPRINT(One)`	erzeugt die Zeichenfolge	`fOnePrint`
`fPRINT(Two)`	erzeugt die Zeichenfolge	`fTwoPrint`
`fPRINT(Three)`	erzeugt die Zeichenfolge	`fThreePrint`

Vordefinierte Makros

Viele Compiler definieren eine Reihe nützlicher Makros.
Folgende Information wird jeweils substituiert:

`__TIME__`	Die aktuelle Zeit beim Compilieren.
`__DATE__`	Das aktuelle Datum wird eingesetzt beim Compilieren.
`__LINE__`	Zeilennummern des Quellcodes.
`__FILE__`	Dateiname des Quellcodes.

assert

Das Makro `assert` liefert `True` zurück, wenn der Parameter zu `True` ausgewertet wird. Ergibt die Auswertung `False`, brechen bestimmte Compiler das Programm ab, andere lösen eine Ausnahme (Exception) aus. Daher eignet sich `assert` zur Fehlersuche.
Ein Vorteil des Makros assert ist, dass der Präprozessor überhaupt keinen Code dafür produziert, wenn `DEBUG` nicht definiert ist.

Anstatt das vom Compiler bereitgestellte `assert` zu benutzen, kann man auch ein eigenes schreiben:

Beispiel

```
// Asserts

#define DEBUG
#include <iostream.h>

#ifndef DEBUG
  #define ASSERT(x)
#else
  #define ASSERT(x) \
          if (!(x)) \
          { \
            cout << "Fehler: " << #x << " nicht zutreffend"; \
            cout << "\nIn Zeile " << __LINE__; \
            cout << "\nIn Datei " << __FILE__; \
          }
#endif
```

```
int main(void)
{
  int x = 5;

  cout << "1. Annahme:\n";
  ASSERT(x==5);

  cout << "\n\n2. Annahme:\n";
  ASSERT(x!=5);

  return(0);
}
```

Ausgabe

```
1. Annahme:

2. Annahme:
Fehler: x!=5 nicht zutreffend
In Zeile 26
In Datei I:\var\CppUebungen\ASSERTS\asserts.cpp
```

Normalerweise werden Makros auf einer Zeile definiert. Hat man aber kompliziertere Makros kann man diese mit dem Backslash-Zeichen (\) auf einer neuen Zeile fortsetzen.

38. try - Blöcke

Codebereiche, die ein Problem hervorrufen können, schliesst man in try-Blöcke ein. Catch-Blöcke folgen unmittelbar auf try-Blöcke um eventuell ausgelöste Exceptions abzufangen, den zugewiesenen Speicher aufzuräumen und den Benutzer in geeigneter Form zu informieren:

```
try
{
  SomeDangerousFunctions();   // Gefährliche Funktionen
}
catch(OutOfMemory)
{
  // Aktionen unternehmen
}
catch(FileNotFound)
{
  // Andere Aktionen unternehmen
}
```