Kapitel C++ als PDF exportieren

C++

C+ + ist eine von der ISO genormte Programmiersprache. Sie wurde ab 1979 von Bjarne Stroustrup bei AT&T als Erweiterung der Programmiersprache C entwickelt. C+ + ermöglicht sowohl die effiziente und maschinennahe Programmierung als auch eine Programmierung auf hohem Abstraktionsniveau. Der Standard definiert auch eine Standardbibliothek, zu der verschiedene Implementierungen existieren.

Lernplattform SoloLearn

- C++ spielerisch lernen

Nachschlagewerke

- Einstieg in C++
- WikiBook zur C++ Programmierung
- Kurzanleitung für GNU C/C++ - Compiler
- C++ unter Linux

Behandelte Inhalte

- Bibliothek string.h => C-String
- Bibliothek string => String
- Funktionen
- Parameterübergabe
- Zeiger
- Arrays & Sortieralgorithmen
- Strukturen
- Klassen
- Dateibehandlung

Beispiele vom Unterricht

- Beispiel zur Parameterübergabe bei Funktionen
- Beispiel zu Arrays, Schleifen und Funktionen
- Beispiel zu Arrays, Funktionen, Schleifen und Zufallszahlen
- Beispiel zu Arrays, Sortieralgorithmen und Funktionen
- Schiffe versenken einfach (Version 1) - Beispiel zu Mehrdimensionale Arrays, Funktionen, Schleifen

- Beispiel zur Dateibehandlung

Die Standard-Bibliothek <string.h> - C-String

Im weiteren Verlauf dieses Buchs werden Sie öfters Funktionen der Headerdatei <string.h> verwenden. Darin sind viele nützliche Funktionen enthalten, die die Arbeit mit Char-Arrays (=C-Strings) vereinfachen.

strcpy() - char Arrays kopieren

Wollen Sie ein char-Array in ein anderes char-Array kopieren, können Sie die Funktion strcpy() (string copy) nutzen. Die Syntax lautet:

```
char *strcpy(char *s1, char *s2);
```

Dass hierbei das char-Array s1 groß genug sein muss, versteht sich von selbst. Bitte beachten Sie dabei, dass das Ende-Zeichen '\0' auch Platz in s1 benötigt. Hierzu ein Beispiel:

Beispiel:

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

using namespace std;

int main(void) {

   char ziel_str[50];
   char str1[] = "Das ist ";
   char str2[] = "ein ";
   char str3[] = "Teststring";

   strcpy(ziel_str, str1);
   /* Ein umständliches Negativbeispiel */
   strcpy(&ziel_str[8], str2);
   /* So ist es einfacher und sicherer */
   strcat(ziel_str, str3);

   cout << ziel_str;
}
```

Ausgabe:

```
Das ist ein Teststring
```

In diesem Beispiel haben Sie gesehen, dass es auch möglich ist, mit strcpy() Strings aneinanderzuhängen:

```
strcpy(&ziel_str[8], str2);
```

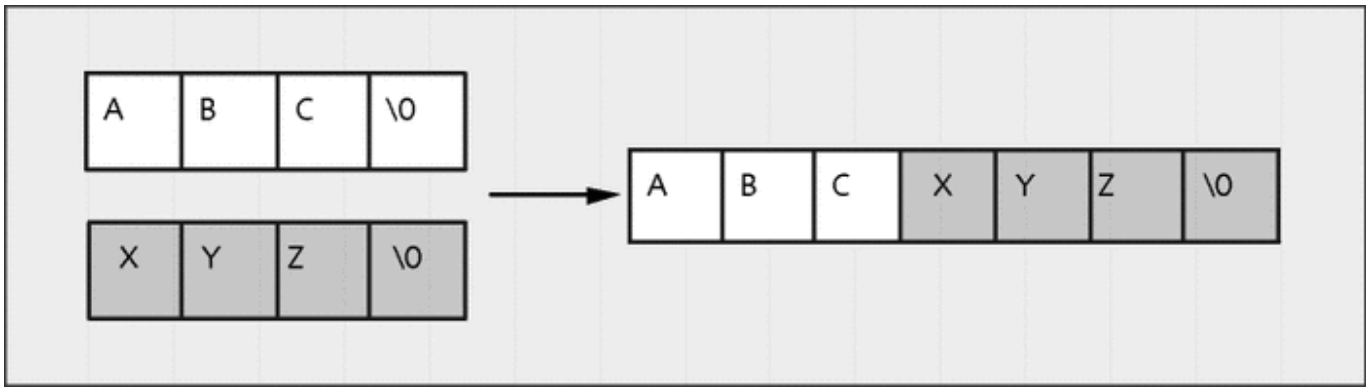
Nur ist das umständlich, und schließlich gibt es dafür die Funktion strcat(). Beim Betrachten der Funktion strcpy() fällt außerdem auf, dass hierbei ebenfalls nicht überprüft wird, wie viele Zeichen in den Zielstring kopiert werden, mit denen wieder auf einen undefinierten Speicherbereich zugegriffen werden kann. Daher ist auch die Funktion strcpy() eine gefährliche Funktion, wenn diese falsch eingesetzt wird.

strcat() – char Arrays aneinanderhängen

Um ein char-Array an ein anderes zu hängen, können Sie die Funktion strcat() (string catenation) verwenden.

```
char *strcat(char *s1, char *s2);
```

Damit wird s2 an das Ende von s1 angehängt, wobei (logischerweise) das Stringende-Zeichen '\0' am Ende von String s1 überschrieben wird. Voraussetzung ist auch, dass der String s2 Platz in s1 hat.



Beispiel:

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

using namespace std;

int main(void) {
   char ziel[30]="ABC\0";
   char name[20]="XYZ\0";

   strcat(ziel, name);
   cout << ziel;

   return 0;
}
```

Ausgabe:

```
ABCXYZ
```

strcmp() - char-Arrays vergleichen

Für das lexikografische Vergleichen zweier char-Arrays kann die Funktion strcmp() verwendet werden. Die Syntax lautet:

```
int strcmp(char *s1, char *s2);
```

Sind beide char-Arrays identisch, gibt diese Funktion 0 zurück. Ist der String s1 kleiner als s2, ist der Rückgabewert kleiner als 0; und ist s1 größer als s2, dann ist der Rückgabewert größer als 0. Ein Beispiel:

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

using namespace std;

void String_Vergleich(char s1[], char s2[]);

int main(void) {
   char str1[] = "aaa";
   char str2[] = "bab";
   char str3[] = "cbb";

   String_Vergleich(str1, str2);
   String_Vergleich(str1, str3);
   String_Vergleich(str3, str2);
   String_Vergleich(str1, str1);

   return 0;
}

void String_Vergleich(char s1[], char s2[]) {
    int ret = strcmp (s1, s2);
    if(ret == 0)
    {
        cout << s1 << " == " << s2 << endl;
    }
    else if(ret < 0)
    {
        cout << s1 << " < " << s2 << " --> das erste ungleiche Zeichen in s1
ist kleiner als in s2!" << endl;
    }
    else {
        cout << s1 << " > " << s2 << " --> das erste ungleiche Zeichen in s1
ist groesser als in s2!" << endl;
    }
}
```

Ausgabe:

```
aaa < bab --> das erste ungleiche Zeichen in s1 ist kleiner als in s2!
aaa < cbb --> das erste ungleiche Zeichen in s1 ist kleiner als in s2!
cbb > bab --> das erste ungleiche Zeichen in s1 ist groesser als in s2!
aaa == aaa
```

strlen() – Länge eines char-Arrays ermitteln

Um die Länge eines Strings zu ermitteln, kann die Funktion strlen() (string length) eingesetzt werden. Die Syntax lautet:

```
size_t strlen(char *s1);
```

Damit wird die Länge des adressierten char-Arrays s1 ohne das abschließende Stringende-Zeichen zurückgegeben. Das Beispiel zu strlen():

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

using namespace std;

int main(void) {
   char string[] = "Das ist ein Teststring";
   size_t laenge;

   laenge = strlen(string);
   cout << laenge << endl;

   char teststring[] ="Das\0 ist ein Test";
   cout << strlen(teststring) << endl;

   return 0;
}
```

HINWEIS: Dass die Funktion strlen() das Stringende-Zeichen '\0' nicht mitzählt, ist eine häufige Fehlerquelle, wenn es darum geht, dynamisch Speicher für einen String zu reservieren. Denken Sie daran, dass Sie immer Platz für ein Zeichen mehr bereithalten.

Ausgabe:

```
22
3
```

Weitere char-Array Funktionen

siehe

char-Array Funktionen

Die Bibliothek string

Die von C geerbten Zeichenketten haben den großen Nachteil, dass der Speicher nicht dynamisch nach Anforderung belegt wird, sondern dass immer so viel Speicher blockiert wird, wie im schlechtesten Fall erforderlich ist. Auch das Fehlerpotenzial, das durch das Fehlen der Abschluss-Null und der fehlenden Kontrolle der Speichergrenzen entsteht, hat viele Programmierer veranlasst, eine eigene Stringklasse zu schreiben. Inzwischen ist aber eine Klasse string in der Standardbibliothek verfügbar.

Verwendung

Um ein Objekt vom Typ string zu definieren, muss zunächst die Datei string eingebunden werden. Sie endet wie viele Header-Dateien der Standardbibliothek von C + + nicht auf .h. Die Klasse string liegt im Namensraum std. Sofern Sie nicht jedes Mal std::string angeben wollen, um einen String zu definieren, ziehen Sie zu Anfang den Namensraum std mit dem Befehl using hinzu:

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

int main (void) {
  string meinName;
  return 0;
}
```

Kompatibilität

Ein Objekt vom Typ string verhält sich in vielen Dingen kompatibel zum char-Array. Der Zugriff auf die einzelnen Zeichen eines Strings erfolgt wie beim klassischen C-String durch die eckigen Klammern. Es ist möglich, Strings mit C-Strings zu initialisieren. Sie können sie ineinander umformen und miteinander vergleichen.

Unterschiede

Der Hauptunterschied besteht darin, dass Sie einen String nicht über einen char-Zeiger manipulieren können, wie Sie das mit einem char-Array gemacht haben. Ein Objekt der Klasse string besteht ja nicht nur aus den Zeichen selbst. Wenn Sie also einen Zeiger auf den String verwenden, zeigt dieser nicht auf den ersten Buchstaben, sondern auf das Objekt, das die Zeichen verwaltet. Da das Objekt in der Lage sein muss, die Daten an eine andere Stelle zu kopieren, wenn die Zeichenkette größer wird, ist es gar nicht ratsam, mit einem Zeiger auf die Zeichen direkt zuzugreifen. Stattdessen bietet die Klasse eigene Zeiger an, die »Iteratoren« genannt werden. Sie erfordern allerdings eine besondere Handhabung. Wir werden darauf noch zurück kommen. Das Fehlen der direkten Zugriffe schränkt den Programmierer zwar ein, erhöht aber die Sicherheit des Programms.

Vorteile

Die Klasse string bietet gegenüber den klassischen C-Strings einige Vorteile. So muss beim Anlegen eines Objekts nicht angegeben werden, wie viele Zeichen reserviert werden sollen. Wird mehr Platz benötigt, sorgt das Objekt selbst dafür, dass es den erforderlichen Speicherbereich bekommt. Der größte Vorteil ist vielleicht, dass Sie Strings einfach zuweisen und vergleichen können, wie Sie es von Integer-Variablen gewöhnt sind.

Zuweisung

Ein String kann durch eine Zuweisung direkt in eine andere Stringvariable kopiert werden. Sie brauchen also keine spezielle Funktion strcpy(), müssen auch keine Schleife über die Elemente machen, nicht darüber nachdenken, ob die Abschluss-Null gesetzt ist, und nicht kontrollieren, ob der Zielspeicher für eine Kopie überhaupt ausreicht. Sie können dem string-Objekt sogar ohne eine explizite Konvertierung einen klassischen C-String zuweisen.

Verbinden

Strings können aneinander gehängt werden, indem ein einfaches Pluszeichen zwischen sie gestellt wird.

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


int main (void)
{
    string meinName = "neu";
    char oldName[25] = "alt";
    string neuString;


    neuString = meinName;            // "neu"
    neuString = oldName;             // "alt"
    neuString = meinName + oldName;  // "neualt"
    neuString += oldName;            // "neualtalt"

    return 0;
}
```

Vergleiche

Im Gegensatz zu den C-Strings können für Abfragen jetzt die einfachen Operatoren verwendet werden, die bereits von den Zahlenvergleichen bekannt sind. Es sind also keine speziellen Funktionen

wie strcmp() mehr erforderlich.

Operator	Bedeutung
==	gleich
!=	ungleich
<	in lexikalischer Reihenfolge vorher
>	in lexikalischer Reihenfolge nachher
< =	in lexikalischer Reihenfolge vorher oder gleich
> =	in lexikalischer Reihenfolge nachher oder gleich

Für den Vergleich reicht es, dass einer der beiden Operanden vom Typ string ist. Er kann sowohl mit einem klassischen C-String als auch mit einem char verglichen werden. Die Reihenfolge spielt keine Rolle.

```
string s;
char cstring[256];
    ...
    if (s < cstring)
    ...
    if (cstring < s)
```

Elementfunktionen

Darüber hinaus bringt die Klasse string eine Reihe anderer Funktionen mit, die in Tabelle (tabstringfunction) aufgeführt sind.

Funktion	Aufgabe
length()	liefert die Länge des Strings
insert(n, s)	fügt den String s an Position n ein
erase(p, n)	Entfernt ab Position p n Zeichen
find(s)	Liefert die Position, an der sich der String s befindet
substr(pos1, pos2)	Gibt den Substring von der Position 1 bis zur Position 2 zurück

Beispiel:

Im folgenden Code-Ausschnitt sind die Funktionen noch einmal in einem Programm dargestellt. In den Kommentaren stehen die Ergebnisse der Funktion.

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

int main()
{
    string s("123");    // Konstruktor fuer C-String
    len = s.length();   // liefert 3, die Laenge des Strings
    s.insert(2, "xy");  // s ist nun "12xy3"
    s.erase(2,2);       // s ist nun "123"
    pos = s.find("23"); // liefert die Position 1
```

```
    s.substr(0,2);      // liefert den Substring von der Position 0 bis 2 -
-> 12
}
```

Funktionen in C++

Ein wichtiges Sprachelement von C+ + kam bisher noch überhaupt nicht vor: die Funktion. Die Möglichkeit, Funktionen zu bilden, ist ein herausragendes Merkmal einer Programmiersprache. Ganz allgemein versteht man unter einer Funktion einen in sich geschlossenen Programmteil, der eine bestimmte Aufgabe erfüllt. Der Vorteil einer Funktion ist die Wiederverwendbarkeit.

Betrachten wir folgendes Beispiel:

```
int add( int x, int y)
{
  int z = x+y;

  return z;
}
```

An diesem Codeausschnitt erkennen Sie alle Bestandteile einer Funktion:

- Typ des Rückgabewerts: int
- Funktionsname: add
- Argumentliste: (int x, int y)
- Funktionskörper: Anweisung innerhalb des Blocks der Funktion, begrenzt durch die geschweiften Klammern { und }
- Return-Anweisung: return z; (bei Rückgabetyp void nicht nötig)

Zu all diesen Teilen ist natürlich noch Einiges zu sagen. Vorher noch ein typografischer Hinweis: Es hat sich eingebürgert, in Büchern zwei Klammern hinter Bezeichner zu setzen, die sich auf eine Funktion beziehen, zum Beispiel in Sätzen wie: Mit add() erreichen Sie die Addition zweier Ganzzahlen. Das sagt noch nichts über Art und Umfang der Argumentliste aus, sondern soll Sie lediglich daran erinnern, dass es dabei um eine Funktion und nicht um eine Variable geht. Ich will mich auch in diesem Buch daran halten.

Rückgabewert

In C+ + muss jede Funktion einen Typ für den Wert angeben, den sie zurückliefert. Manchmal ist es aber auch gar nicht nötig oder sinnvoll, dass eine Funktion überhaupt einen Rückgabewert hat. In diesem Fall geben Sie als Typ void an.

Was macht man nun mit einem solchen Wert? Der Programmteil, der die Funktion aufruft, kann diese an allen Stellen einsetzen, wo er sonst eine Variable oder Konstante angeben würde (in obiger Form allerdings nur dort, wo lediglich der Wert benötigt wird), also etwa:

```
int main()
{
  int a = 5;
  int b = 12;
  int c = add(a,b);
```

```
  cout << ``a = `` << a << ``, c = `` << c << ``, a+c = `` << add(a,c) <<
endl;

  return 0;
}
```

Dieses Programm hat dann die Ausgabe: a = 5, c = 17, a+c = 22

Übrigens: Selbst wenn eine Funktion einen Wert zurückgibt, müssen Sie ihn nicht beachten. Sie dürfen auch schreiben:

```
int a = 5;
int b = 12;
add(a,b);
```

auch wenn das hier keinen Sinn machen würde. Bei Funktionen mit Rückgabetyp void ist das hingegen die übliche Form des Aufrufs. Allgemein kommt es aber häufiger vor, dass Rückgabewerte ignoriert werden. Beispielsweise geben viele Funktionen Statusinformationen darüber zurück, wie gut (oder schlecht) sie ihre Aufgabe erfüllen konnten. Viele Anwender solcher Funktionen interessieren sich nicht für den Status und übergehen ihn. Das kann manchmal aber auch gefährlich werden, wenn etwa aufgetretene Fehler aus diesem Grund zunächst unentdeckt bleiben.

Funktionsname

Wie alle anderen Bezeichner in C++ dürfen Sie auch Funktionsnamen nur aus Buchstaben, Ziffern sowie dem Unterstrich _ bilden. Außerdem ist es nicht erlaubt, Funktionsnamen zu verwenden, die Schlüsselwörten gleichen (etwa for).

Argumentliste

Eine Funktion kann immer nur auf den Daten arbeiten, die ihr lokal vorliegen. Außer global (das heißt außerhalb aller Funktionen) definierten Variablen sind das nur die Parameter, die das Hauptprogramm an die Funktion übergibt. Von diesen Parametern (auch Argumente genannt) können Sie keinen, einen oder mehrere angeben, die Sie dann durch Kommas trennen.

Wenn Sie eine Funktion ohne Argumente schreiben wollen, lassen Sie den Bereich zwischen den beiden runden Klammern einfach leer – denn die Klammern müssen Sie stets schreiben! – oder Sie setzen ein Argument vom Typ void ein.

Ansonsten geben Sie für jeden Parameter seinen Datentyp und einen Namen an, unter dem er in der Funktion bekannt sein soll. Dieser Name kann vollkommen anders sein, als der im Hauptprogramm beim Aufruf verwendete. Auch in obigem Beispiel heißen die Summanden in der Funktion x und y, im Hauptprogramm aber a und b.

Funktionskörper

Hier stehen die Anweisungen, die bei einem Aufruf der Funktion ausgeführt werden. Man kann darüber streiten, wie lang Funktionen sein sollten. Es gibt Experten, die fordern, dass eine Funktion aus nicht mehr als 50 Zeilen bestehen dürfe, sonst werde sie unleserlich. Es gibt jedoch in der Praxis immer wieder Fälle, in denen längere Funktionen sinnvoll sind. Bei der objektorientierten Programmierung werden Sie allerdings ohnehin wesentlich mehr Funktionen (beziehungsweise Methoden) verwenden, die im Durchschnitt wesentlich kürzer sind als bei der prozeduralen Programmierung.

Innerhalb des Funktionskörpers können Sie die Funktionsparameter wie normale Variablen verwenden; zusätzlich können Sie natürlich auch noch lokale Variablen definieren. Außerdem ist es selbstverständlich erlaubt, aus einer Funktion wieder andere Funktionen aufzurufen. (Sie dürfen sogar die Funktion selbst wieder aufrufen; man spricht dann von einer rekursiven Funktion – aber das ist ein eigenes Thema.)

Return-Anweisung

Die Anweisungen im Funktionskörper werden so lange abgearbeitet, bis das Programm auf das Ende der Funktion oder eine return-Anweisung trifft. Diese erfüllt einen doppelten Zweck:

Sie legen fest, welchen Wert die Funktion an das Hauptprogramm zurückliefern soll. Das kann eine Variable sein oder ein Ausdruck, eine Konstante oder der Rückgabewert einer anderen Funktion (wobei Letzteres als schlechter Stil gilt). Hat Ihre Funktion den Rückgabetyp void, geben Sie an dieser Stelle überhaupt nichts an. Der Typ des angegebenen Werts muss jedoch in jedem Fall mit dem deklarierten Rückgabetyp übereinstimmen. Sie beenden die Funktion und kehren zum Hauptprogramm zurück. Jede return-Anweisung – sei sie nun am Ende oder irgendwo inmitten des Funktionskörpers – markiert das Ende der Abarbeitung der Funktion und den Rücksprung an die Stelle, von der aus die Funktion aufgerufen wurde. Sie können also die Funktion schon beenden, bevor alle Anweisungen ausgeführt sind, zum Beispiel wenn eine bestimmte Bedingung erfüllt ist. Ist der Rückgabetyp void, muss am Ende der Funktion keine return-Anweisung stehen (auch nicht das Schlüsselwort return), wie in folgendem Beispiel:

```
void ausgabe( int z)
{
  cout << ``Das Ergebnis ist: `` << z << endl;
}
```

Wenn Sie allerdings bei Funktionen mit irgendeinem anderen Rückgabetyp die return-Anweisung vergessen, meldet der Compiler einen Fehler.

Der Prototyp

Bevor Sie eine Funktion verwenden können, müssen Sie dem Compiler zunächst mitteilen, dass es eine Funktion dieses Namens gibt, wie viele und welche Parameter sie hat und welchen Typ sie zurückliefert. Dies geschieht mit einem so genannten Prototyp der Funktion. Der Prototyp sieht

genauso aus wie die Funktion selbst bis auf den Funktionskörper; dieser fehlt und wird durch ein einfaches Semikolon ; ersetzt. (Es ist sogar erlaubt, die Namen der Argumente wegzulassen und nur ihre Typen anzugeben. Analog zu den Klassen (siehe Seite [*]) ist also der Prototyp die Deklaration und die Funktion mit Körper die Definition.

Aufruf

Der Aufruf einer Funktion erfolgt durch Nennung des Funktionsnamens. An den Funktionsnamen schließt sich immer ein Klammerpaar an, das gegebenenfalls auch Parameter enthalten kann. Dieses Klammerpaar ist zwingend erforderlich. Die Parameter des Aufrufs müssen zu den Parametern der Funktion passen. Besitzt die Funktion einen Rückgabewert, kann der Funktionsaufruf als Ausdruck verwendet werden. Er kann also beispielsweise auf der rechten Seite einer Zuweisung stehen.

Übergabe der Argumente

C+ + kennt mehrere Varianten, wie einer Funktion die Argumente übergeben werden können: call-by-value, call-by-reference und call-by-pointer.

call-by-value (Wertübergabe)

Bei call-by-value (Wertübergabe) wird der Wert des Arguments in einen Speicherbereich kopiert, auf den die Funktion mittels Parametername zugreifen kann. Ein Werteparameter verhält sich wie eine lokale Variable, die „automatisch“ mit dem richtigen Wert initialisiert wird. Der Kopiervorgang kann bei Klassen (Thema eines späteren Kapitels) einen erheblichen Zeit- und Speicheraufwand bedeuten!

Beispiel

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

using namespace std;

//Prototyp, Deklaration
int sumW(int x,int y);   //Parameterübergabe per Wert (Value)

int main(int argc, char** argv) {
    cout << "Hello World" << endl;

    int a=5,b=6,erg=0;
    cout << "Parameteruebergabe per Wert" << endl;
    erg=sumW(a, b);
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    getch();

    return 0;
}

//Funktionendefinition - Parameterübergabe per Wert
int sumW(int x,int y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    return x+y;
```

```
}
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
a	0x0001	5
b	0x0005	6
...	.	.
...	.	.
x	0x00a1	5 → 6
y	0x00a5	6 → 7

Die Ausgabe des Programms ist:

```
Hello World
Parameteruebergabe per Wert
x: 6
y: 7
a: 5
b: 6
```

call-by-reference (Übergabe per Referenz)

Sollen die von einer Funktion vorgenommen Änderungen auch für das Hauptprogramm sichtbar sein, müssen in C sogenannte Zeiger verwendet werden. C+ + stellt ebenfalls Zeiger zur Verfügung. C+ + gibt Ihnen aber auch die Möglichkeit, diese Zeiger mittels Referenzen zu umgehen, was im alten C nicht möglich war. Beide sind jedoch noch Thema eines späteren Kapitels.

Im Gegensatz zu call-by-value wird bei call-by-reference die Speicheradresse des Arguments übergeben, also der Wert nicht kopiert. Änderungen der (Referenz-)Variable betreffen zwangsläufig auch die übergebene Variable selbst und bleiben nach dem Funktionsaufruf erhalten. Um call-by-reference anzuzeigen, wird der Operator & verwendet, wie Sie gleich im Beispiel sehen werden. Wird keine Änderung des Inhalts gewünscht, sollten Sie den Referenzparameter als const deklarieren, um so den Speicherbereich vor Änderungen zu schützen. Fehler, die sich aus der ungewollten Änderung des Inhaltes einer übergebenen Referenz ergeben, sind in der Regel schwer zu finden.

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

using namespace std;

//Prototyp, Deklaration
int sumR(int &x,int &y);   //Parameterübergabe per Referenz (Reference)


int main(int argc, char** argv) {
    cout << "Hello World" << endl;
```

```
    cout << "Parameteruebergabe per Referenz" << endl;
    a=5,b=6,erg=0;
    erg=sumR(a, b);
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;


    getch();

    return 0;
}

//Funktionendefinition - Parameterübergabe per Referenz
int sumR(int &x,int &y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    return x+y;
}
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
x,a	0x0001	5 → 6
y,b	0x0005	6 → 7
...	.	
...	.	
...	.	
...	.	

Die Ausgabe des Programms ist:

```
Hello World
Parameteruebergabe per Referenz
x: 6
y: 7
a: 6
b: 7
```

call-by-pointer (Übergabe per Zeiger)

Wenn Sie einen Zeiger als Parameter an eine Funktion übergeben, können Sie den Wert an der übergebenen Adresse ändern.

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

using namespace std;

//Prototyp, Deklaration
int sumZ(int *x,int *y);   //Parameterübergabe per Zeiger (Pointer)

int main(int argc, char** argv) {
    cout << "Hello World" << endl;
    cout << "Parameteruebergabe per Zeiger" << endl;
    a=5,b=6,erg=0;
    erg=sumZ(&a, &b);
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    getch();

    return 0;
}


//Funktionendefinition - Parameterübergabe per Zeiger
int sumZ(int *x,int *y){
    *x+=1; //x=x+1; x++; ++x;
    *y+=1;

    cout << "x: " << *x <<endl;
    cout << "y: " << *y <<endl;

    return *x+*y;
}
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
a	0x0001	5 → 6
b	0x0005	6 → 7
...	.	
...	.	
*x	0x00a1	0x0001
*y	0x00a5	0x0005

Die Ausgabe des Programms ist:

```
Hello World
Parameteruebergabe per Zeiger
x: 6
y: 7
```

```
a: 6
b: 7
```

Funktionen - Beispiel

Angabe

Dieses Programm zeigt die verschiedenen Arten der Parameterübergabe bei Funktionen auf. Es werden jeweils dieselben Variablen auf 3 verschiedenen Arten (per Wert, per Referenz, per Zeiger) übergeben.

Lösung

```
/* Beispiel: Funktionen - Parameterübergabe
   Filename:Parameteruebergabe.cpp
   Author:Lahmer
   Title:Parameterübergabe
   Description: Anhand dieses Beispiels werden die verschiedenen Arten der
Parameterübergabe erklärt
   Last Change:16.01.2018
*/

//Hedader-Dateien bzw. Bibliotheken
#include <iostream>
#include <conio.h>

//Namespace (Namensraum)
using namespace std;

//Prototyp, Deklaration
int sumW(int x,int y);   //Parameterübergabe per Wert (Value)
int sumR(int &x,int &y);   //Parameterübergabe per Referenz (Reference)
int sumZ(int *x,int *y);   //Parameterübergabe per Zeiger (Pointer)


//Hauptprogramm
int main(int argc, char** argv) {
    //Ausgabe
    cout << "Hello World" << endl;

    //Lokale Variablen mit Datentyp int => ganzzahlige Zahlen
    int a=5,b=6,erg=0;

    //Parameterübergabe per Wert
    cout << "Parameteruebergabe per Wert" << endl;
    //Funktionsaufruf per Wert = Kopie der Variable
    erg=sumW(a, b);
    //Ausgabe der Variablen a bzw. b
    cout << "a: " << a <<endl;
```

```
    cout << "b: " << b <<endl;

    //Parameterübergabe per Referenz
    cout << "Parameteruebergabe per Referenz" << endl;
    a=5,b=6,erg=0;
    //Funktionsaufruf per Referenz = Verknüpfung der Variablen a bzw. b
    erg=sumR(a, b);
    //Ausgabe der Variablen a bzw. b
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    //Parameterübergabe per Zeiger
    cout << "Parameteruebergabe per Zeiger" << endl;
    a=5,b=6,erg=0;
    //Funktionsaufruf per Referenz = Übergabe der Adressen der Variablen a
bzw. b
    erg=sumZ(&a, &b);
        //Ausgabe der Variablen a bzw. b
    cout << "a: " << a <<endl;
    cout << "b: " << b <<endl;

    //Auf Tastendruck warten
    getch();

    //Rückgabewert = 0
    return 0;
}

//Funktionendefinition - Parameterübergabe per Wert
int sumW(int x,int y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    //Rückgabewert Summe von x+y
    return x+y;
}

//Funktionendefinition - Parameterübergabe per Referenz
int sumR(int &x,int &y){
    x+=1; //x=x+1; x++; ++x;
    y+=1;

    cout << "x: " << x <<endl;
    cout << "y: " << y <<endl;

    //Rückgabewert Summe von x+y
    return x+y;
```

```
}

//Funktionendefinition - Parameterübergabe per Zeiger
int sumZ(int *x,int *y){
    *x+=1; //x=x+1; x++; ++x;
    *y+=1;

    cout << "x: " << *x <<endl;
    cout << "y: " << *y <<endl;

    //Rückgabewert Summe von x+y
    return *x+*y;
}
```

Ausgabe

```
Hello World
Parameteruebergabe per Wert
x: 6
y: 7
a: 5
b: 6
Parameteruebergabe per Referenz
x: 6
y: 7
a: 6
b: 7
Parameteruebergabe per Zeiger
x: 6
y: 7
a: 6
b: 7
```

Zeiger (Pointer)

Zeiger (engl. pointers) sind Variablen, die als Wert die Speicheradresse einer anderen Variable enthalten.

Jede Variable wird in C++ an einer bestimmten Position im Hauptspeicher abgelegt. Diese Position nennt man Speicheradresse (engl. memory address). C++ bietet die Möglichkeit, die Adresse jeder Variable zu ermitteln. Solange eine Variable gültig ist, bleibt sie an ein und derselben Stelle im Speicher.

Am einfachsten vergegenwärtigt man sich dieses Konzept anhand der globalen Variablen. Diese werden außerhalb aller Funktionen und Klassen deklariert und sind überall gültig. Auf sie kann man von jeder Klasse und jeder Funktion aus zugreifen. Über globale Variablen ist bereits zur Kompilierzeit bekannt, wo sie sich innerhalb des Speichers befinden (also kennt das Programm ihre Adresse).

Zeiger sind nichts anderes als normale Variablen. Sie werden deklariert (und definiert), besitzen einen Gültigkeitsbereich, eine Adresse und einen Wert. Dieser Wert, der Inhalt der Zeigervariable, ist aber nicht wie in unseren bisherigen Beispielen eine Zahl, sondern die Adresse einer anderen Variable oder eines Speicherbereichs. Bei der Deklaration einer Zeigervariable wird der Typ der Variable festgelegt, auf den sie verweisen soll.

```
#include <iostream>

int main() {
     int   Wert;      // eine int-Variable
     int *pWert;      // eine Zeigervariable, zeigt auf einen int
     int *pZahl;      // ein weiterer "Zeiger auf int"

     Wert = 10;       // Zuweisung eines Wertes an eine int-Variable

     pWert = &Wert;   // Adressoperator '&' liefert die Adresse einer
Variable
     pZahl = pWert;   // pZahl und pWert zeigen jetzt auf dieselbe Variable
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
Wert	0x0001	10
*pWert	0x0005	0x0001
*pZahl	0x0009	0x0005
...	.	.
...	.	.

Der Adressoperator & kann auf jede Variable angewandt werden und liefert deren Adresse, die man einer (dem Variablentyp entsprechenden) Zeigervariablen zuweisen kann. Wie im Beispiel gezeigt, können Zeiger gleichen Typs einander zugewiesen werden. Zeiger verschiedenen Typs bedürfen einer Typumwandlung. Die Zeigervariablen pWert und pZahl sind an verschiedenen Stellen im Speicher abgelegt, nur die Inhalte sind gleich.

Wollen Sie auf den Wert zugreifen, der sich hinter der im Zeiger gespeicherten Adresse verbirgt, so verwenden Sie den Dereferenzierungsoperator *.

```
  *pWert += 5;
  *pZahl += 8;

   std::cout << "Wert = " << Wert << std::endl;
```

Beispielhafte Speicherbelegung des Programms im Hauptspeicher:

Variable	Adresse	Wert
Wert	0x0001	10 –> 15 –> 23
*pWert	0x0005	0x0001
*pZahl	0x0009	0x0005
...	.	.
...	.	.

Ausgabe:

```
Wert = 23
```

Man nennt das den Zeiger dereferenzieren. Im Beispiel erhalten Sie die Ausgabe Wert = 23, denn pWert und pZahl verweisen ja beide auf die Variable Wert.

Um es noch einmal hervorzuheben: Zeiger auf Integer (int) sind selbst keine Integer. Den Versuch, einer Zeigervariablen eine Zahl zuzuweisen, beantwortet der Compiler mit einer Fehlermeldung oder mindestens einer Warnung. Hier gibt es nur eine Ausnahme: die Zahl 0 darf jedem beliebigen Zeiger zugewiesen werden. Ein solcher Nullzeiger zeigt nirgendwohin. Der Versuch, ihn zu dereferenzieren, führt zu einem Laufzeitfehler.

Zeiger- Beispiel

Angabe / Beschreibung

Dieses Programm demonstriert die Anwendung von Zeigern. Weiters werden auch wieder Variablen per Wert, Referenz und Zeiger an Funktionen übergeben.

Lösung

```
/* Beispiel: Zeiger
   Filename:main.cpp
   Author:Lahmer
   Title:Verdeutlichung von Zeiger
   Description: Anhand dieses Beispiels werden Zeiger näher erläutert
   Last Change:16.01.2018
*/


//Hedader-Dateien bzw. Bibliotheken
#include <iostream>
#include <conio.h>

//Namespace
using namespace std;


//Funktionsprototyp
int diff(int a, int b); //Parameterübergabe per Wert => Rückgabewert int, 2 int-Übergabeparameter
void erhoehezahl(int &a); //Parameterübergabe per Referenz => Rückgabewert void (nichts), Variable i wird als Referenz übergeben
void verringerezahl(int *b); //Parameterübergabe per Zeiger => Rückgabewert void (nichts), Variable b wird als Zeiger übergeben


//Hauptprogramm
int main(int argc, char** argv) {

    //Lokale Variablendeklaration
    int z1=99,z2=23;
    int erg=0;
    int *ptr;
    int *ptr2;    //ptr2,ptr soll auf z2 zeigen: ptr2, ptr --> z2;
    int *ptr3;    //ptrx soll auf z1 zeigen: ptrx --> z1
```

```
    //Ausgabe der Speicheradressen der angelegten Variablen
    cout << "Adresse der Variable z1: " << &z1 << endl;
    cout << "Adresse der Variable z2: " << &z2 << endl;
    cout << "Adresse der Variable erg: " << &erg << endl;
    cout << "Adresse der Variable ptr: " << &ptr << endl;
    cout << "Adresse der Variable ptr2: " << &ptr2 << endl;
    cout << "Adresse der Variable ptr3: " << &ptr3 << endl;
    cout << endl << endl;


    //Die Adresse von der Variable z2 wird in die Variable ptr gespeichert
    ptr=&z2;

    //Der Inhalt von ptr (also die Adresse von z2) wird in die Variable ptr2
gespeichert
    ptr2=ptr;

    //Die Adresse von z1 wird in die Variable ptr3 gespeichert
    ptr3=&z1;


    cout << "Pointer1: " << endl;
    cout << &ptr << endl;    //Ausgabe der Adresse der Variable ptr
    cout << ptr << endl;    //Ausgabe des Inhalts von ptr (also die Adresse
von z2)
    cout << *ptr << endl;    //Ausgabe des Inhalts von der in ptr
gespeicherten Adresse (also der Inhalt von z2)

    cout << "Pointer2: " << endl;
    cout << &ptr2 << endl;    //Ausgabe der Adresse der Variable ptr2 (also
die Adresse von ptr2)
    cout << ptr2 << endl;    //Ausgabe der Variable ptr2 (also die Adresse
von z2)
    cout << *ptr2 << endl << endl;    //Ausgabe des Inhalts an der
gespeicherten Adresse in ptr2 (also der Inhalt von z2)

    cout << "Pointer3: " << endl;
    cout << &ptr3 << endl;    //Ausgabe der Adresse der Variable ptr3
    cout << ptr3 << endl;    //Ausgabe der Variable ptr3 (also die Adresse
von z1)
    cout << *ptr3 << endl << endl;     //Ausgabe des Inhalts an der
gespeicherten Adresse in ptr3 (also der Inhalt von z1)


    cout << "########################" << endl;


    //Funktionsaufrufe
    //Die Variablen z1 und z2 werden per Wert übergeben (=Kopie im Speicher)
    erg=diff(z1,z2);
    cout << erg << endl;
```

```
    //Die Variable z1 wird per Referenz übergeben, d.h. eine neue zweite Variable in der Funktion erhoehezahl() zeigt auf denselben Speicherplatz von z1
    erhoehezahl(z1);
    cout << z1 << endl;

    //Die Variable z2 wird per Zeiger übergeben, d.h. die Adresse von z2 wird an die Funktion verringerezahl() übergeben, die Variable b in der Funktion enthält dann die Adresse von z2
    verringerezahl(&z2);
    cout << z2 << endl;

    *ptr = *ptr*10;  //Der Inhalt an der gespeicherten Adresse in ptr (also z2) wird mit 10 multipliziert und an den Inhalt der in ptr gespeicherten Adresse (also z2) geschrieben
    *ptr3 = *ptr2;   //Der Inhalt an der gespeicherten Adresse in ptr2 (also z2) wird über den Inhalt an der gespeicherten Adresse in ptr3 geschrieben

    cout << endl << endl;
    cout << "Inhalt der Variable z1: " << z1 << endl;
    cout << "Inhalt der Variable z2: " << z2 << endl;


    //Rückgabewert = 0
    return 0;
}

//Funktionsdefinition
int diff(int a, int b)
{
    //Rückgabe der Differenz a-b
    return a-b;
}


void erhoehezahl(int &a)
{
    //Die Variable a wird um 1 erhöht, Aufgrund der Übergabe der Referenz wird auch gleichzeitig die Variable z1 im Hauptprogramm erhöht
    a++;
}

void verringerezahl(int *b)
{
    //Der Inhalt der in der Variable b gespeicherten Adresse wird um 1 verringert (d.h. z2 wird verringert)
    --*b;
    //Der Inhalt der in der Variable b gespeicherten Adresse wird um 1 verringert (d.h. z2 wird verringert)
```

```
    (*b)--;
    //Der Inhalt der in der Variable b gespeicherten Adresse wird um 1
verringert (d.h. z2 wird verringert)
    *b-=1;
}
```

Beispielhafter Auszug des Speichers

Variable	Adresse	Inhalt
z1	0x9ffe3c	99 → 100 → 200
z2	0x9ffe38	23 → 22 → 21 → 20 → 200
erg	0x9ffe34	0 → 76
ptr	0x9ffe28	0x9ffe38
ptr2	0x9ffe20	0x9ffe38
ptr3	0x9ffe18	0x9ffe3c

Ausgabe des Programms

```
Adresse der Variable z1: 0x9ffe3c
Adresse der Variable z2: 0x9ffe38
Adresse der Variable erg: 0x9ffe34
Adresse der Variable ptr: 0x9ffe28
Adresse der Variable ptr2: 0x9ffe20
Adresse der Variable ptr3: 0x9ffe18


Pointer1:
0x9ffe28
0x9ffe38
23
Pointer2:
0x9ffe20
0x9ffe38
23

Pointer3:
0x9ffe18
0x9ffe3c
99

########################
76
100
20


Inhalt der Variable z1: 200
Inhalt der Variable z2: 200
```

```
--------------------------------
Process exited after 0.07103 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Arrays

Eindimensionale Arrays

Arrays und Funktionen

Einfache Sortieralgorithmen

Sortieralgorithmen

Mehrdimensionale Arrays

Mehrdimensionale Arrays

Eindimensionale Arrays und Funktionen

Beispiel: Ermittlung von n Quicktipps im Lotto 6 aus 45 Wir beginnen mit einer Version ohne und Funktionen und bauen diese dann entsprechend um!

```
void main()
{
  int n;  //Anzahl der Tipps
  int tipp[6];
  srand(time(0));
  cout<<"Wie viele Tipps moechten Sie erhalten?";
  cin>>n;
  for(int i=1;i<=n;i++){     //n Tipps ermitteln
    cout<<"\n"<<i<<". Tipp: ";

    for(int j=0; j<6;j++){
        tipp[j]=rand()%45+1;
        cout<<tipp[j]<<" ";
    }
  }
  getch();
}
```

Bei der Verwendung von Arrays in Funktionen sind einige Besonderheiten zu beachten:

- Eine Funktion kann keinen Array-Typ besitzen.
- Werden Arrays als Parameter verwendet, so sind dies in C++ automatisch Ein-Ausgabe-Parameter. Dh. es wird im Unterprogramm keine Kopie der Array-Variablen angelegt, sondern das Unterprogramm greift direkt auf die Array-Variable der aufrufenden Funktion zu.

Wir erweitern nun das obige Beispiel um den Einsatz von Funktionen:

```
void erzeugeQuicktipp(int a[6]);

void main()
{
  int n;  //Anzahl der Tipps
  int tipp[6];
  srand(time(0));
  cout<<"Wie viele Tipps moechten Sie erhalten?";
  cin>>n;
  for(int i=1;i<=n;i++){     //n Tipps ermitteln
    cout<<"\n"<<i<<". Tipp: ";
    erzeugeQuicktipp(tipp);
  }
  getch();
}

void erzeugeQuicktipp(int a[6]){
```

```
  for (int i=0; i<6; i++) {
      a[i]=rand()%45+1;
      cout<<a[i]<<" ";
  }
}
```

Erweitere das Beispiel nun um eine eingene Funktion für die Ausgabe. Außerdem soll verhindert werden, dass innerhalb eines Tipps Zahlen doppelt vorkommen.

```
    // Programm: lotto.cpp
    // Beschreibung: Erzeugen von Lotto-Tips
    #include <iostream>      // Zusatzbibliothek für Ein-und Ausgaben wird
eingebunden
    #include <conio.h>       // Zusatzbibliothek für Konsole wird
eingebunden
    #include <stdlib.h>      // Wird für Zufallszahl-Generator benötigt
    #include <time.h>        // Wird für Initialisierung des Zufallszahl-
Generator benötigt
    using namespace std;     // Standard-Namensraum wird eingestellt

    void erzeugeQuicktipp(int a[6]);
    void ausgabe(int a[6]);

    int main(){
      int n;  //Anzahl der Tipps
      int tipp[6];
      srand(time(0));
      cout<<"Wie viele Tipps moechten Sie erhalten?"; cin>>n;
      for(int i=1;i<=n;i++){     //n Tipps ermitteln
        cout<<"\n"<<i<<". Tipp: ";
        erzeugeQuicktipp(tipp);
        ausgabe(tipp);
      }
      getch();
      return 0;
    }

    void erzeugeQuicktipp(int a[6]){
      for (int i=0; i<6; i++) {
        a[i]=rand()%45+1;
        //Überprüfe, ob Zahl bereits vorhanden
        for(int j=0;j<i;j++){   //Durchlaufe die vorhandenen Einträge
           if(a[i]==a[j]){  //und überprüfe sie auf Gleichheit mit dem
aktuellen Eintrag
            i--;        //ist die Zahl bereits vorhanden, so setze
                    //Zähler i zurück, damit diese Zufallszahl
                        //automatisch nochmals erzeugt wird!
           }
        }
      }
    }
```

```
void ausgabe(int a[6]){
  for (int i=0; i<6; i++) {
     cout<<a[i]<<" ";
  }
}
```

Aufgaben

- Aufgaben

Arrays, Schleifen und Funktionen - Beispiel

Angabe / Beschreibung

Der Benutzer soll eine beliebige ganzzahlige Zahl eingeben und das Programm prüft, ob diese Zahl eine Primzahl ist. Wenn ja, wird die Primzahl in einem Feld der Größe 10 gespeichert.

Lösung

```
/* Beispiel: Arrays und Schleifen
   Filename: main.cpp
   Author: Lahmer
   Title: Primzahlenüberprüfung
   Description: Der Benutzer soll eine beliebige ganzzahlige Zahl eingeben
und das Programm prüft, ob diese Zahl eine Primzahl ist. Wenn ja, wird die
Primzahl in einem Feld der Größe 10 gespeichert.
   Last Change:16.01.2018
*/
//Header-Dateien
#include <iostream>
#include <conio.h>

//Namespace
using namespace std;

//Funktionsprototyp
bool checkPrimzahl(int z);  //Rückgabewert bool, Übergabe per Wert ->
Variable z
void ausgabe(int array[10],int anzahl);   //Rückgabewert void (nichts),
Übergabe per Zeiger (array, Arrays werden immer per Zeiger übergeben) und
Übergabe per Wert (anzahl)


//Hauptprogramm
int main(int argc, char** argv) {

    //Lokale Variablendeklaration
    //Integer Array
    int feld[10];
    bool erg;
    int zahl;
    char nochmal=' ';
    int k=0;

    //Beginn der Do-While Schleife
    do {
```

```
        cout << "Geben Sie bitte eine ganzzahlige Zahl ein: ";
        cin >> zahl; //Eingabe der Zahl

        //Aufruf der Funktion checkPrimzahl
        erg=checkPrimzahl(zahl);
        cout << endl << endl;

        //Bedinung, Prüfen ob erg == true ist, wenn ja => Zahl ist Primzahl,
wenn nein => Zahl ist keine Primzahl
        if(erg==true)
        {
            cout << "Die eingegebene Zahl ist EINE Primzahl!" << endl;
            feld[k]=zahl;     //Zahl wird in das Array namens feld an die
Stelle k gespeichert
            k++;    //k wird um 1 erhöht
        }
        else
        {
            cout << "Die eingegebene Zahl ist KEINE Primzahl" << endl;
        }
        cout << "Willst du noch eine ganzzahlige Zahl eingeben? (j/n)" <<
endl;
        //Einlesen der Entscheidung, ob der Benutzer nochmals eine Zahl
eingeben will
        cin >> nochmal;
        cout << endl << "##################################################"
<< endl;

    } while (nochmal=='j');  //Bedingung der do-while Schleife => Solange
der Benutzer j eingibt, wird das Programm wiederholt

    //Alle Primzahlen gesammelt ausgeben
    ausgabe(feld, k);    //Übergabe der Adresse von feld und der Anzahl der
Primzahlen (k)

    return 0;
}

//Funktionsdefinition
bool checkPrimzahl(int z)
{
    //Lokale Variablendeklaration bool = kann nur true oder false beinhalten
    bool primZ=true;

    //Prüfe alle Teiler von 2 bis z-1
    for(int i=2; i<z; i++)
    {
        //Testen ob z/i ohne Rest teilbar
        if(z%i==0)
        {
```

```
            primZ=false;
        }
    }

    //Rückgabe des booleans ob die Zahl ein Primzahl ist (true oder false)
    return primZ;
}


void ausgabe(int array[10], int anzahl)
{
    //Ausgabe aller Primzahlen
    for(int i=0; i<anzahl; i++)
    {
        cout << array[i] << endl;
    }
}
```

Ausgabe

```
Geben Sie bitte eine ganzzahlige Zahl ein: 16


Die eingegebene Zahl ist KEINE Primzahl
Willst du noch eine ganzzahlige Zahl eingeben? (j/n)
j

##################################################
Geben Sie bitte eine ganzzahlige Zahl ein: 13


Die eingegebene Zahl ist EINE Primzahl!
Willst du noch eine ganzzahlige Zahl eingeben? (j/n)
j

##################################################
Geben Sie bitte eine ganzzahlige Zahl ein: 317


Die eingegebene Zahl ist EINE Primzahl!
Willst du noch eine ganzzahlige Zahl eingeben? (j/n)
n

##################################################
13
317

--------------------------------
Process exited after 19.88 seconds with return value 0
```

```
Drücken Sie eine beliebige Taste . . .
```

Arrays, Funktionen, Schleifen und Zufallszahlen - Beispiel

Angabe / Beschreibung

Der Benutzer gibt die Anzahl der zu erstellenden Zufallszahlen ein und das Programm generiert die Zufallszahlen (zw. 1991 und 2018) und speichert diese in ein Array. Danach wird der Mittelwert und die Summe der zufällig generierten Zahlen berechnet.

Lösung

```
/* Beispiel: Arrays, Funktionen, Schleifen und Zufallszahlen
   Filename: main.cpp
   Author: Lahmer
   Title: Berechnung des Mittelwerts und der Summe von Zufallszahlen
   Description: Der Benutzer gibt die Anzahl der zu erstellenden
Zufallszahlen ein und das Programm generiert die Zufallszahlen (zw. 1991 und
2018) und speichert diese in ein Array. Danach wird der Mittelwert und die
Summe der zufällig generierten Zahlen berechnet.
   Last Change:16.01.2018
*/
//Header-Dateien
#include <iostream>
#include <stdlib.h>
#include <time.h>

//Namensraum
using namespace std;

//Funktionsdeklaration
//Funktion, die das Array mit Zufallswerten befüllt
void erstelleArray(int *feld, int anzahl);    //äquivalent zu void
erstelleArray(int feld[]); -> Rückgabewert void (nichts), Parameterübergabe
per Pointer (feld) und per Wert (anzahl)
void ausgabe(int feld[], int anzahl);  //Rückgabewert void (nichts),
Parameterübergabe per Pointer (feld) und per Wert (anzahl)
int summe(int feld[], int anzahl); //Rückgabewert int (ganzzahlige Zahl),
Parameterübergabe per Pointer (feld) und per Wert (anzahl)
float mittelwert(int summe, int anzahl); //Rückgabewert float
(Gleitkommazahl), Parameterübergabe per Wert (summe, anzahl)


//Hauptprogramm
int main(int argc, char** argv) {
```

```
    //Lokale Variablendeklaration
    int anz=0;

    cout << "Geben Sie bitte die Anzahl der Werte des Arrays an: ";
    //Einlesen einer ganzzahligen Zahl
    cin >> anz;

    //Lokale Variablendeklaration
    int array[anz];        //Deklaration eines Arrays namens array mit der
Größe anz
    int sum=0;
    float mittel=0.0;    //Deklaration der Variable mittel und
Initialisierung mit 0.0

    //Funktionsaufruf
    erstelleArray(&array[0],anz);   //äquivalent zu
erstelleArray(feld,anzahl);
    ausgabe(&array[0],anz);
    sum=summe(&array[0], anz);
    mittel=mittelwert(sum, anz);

    //Ausgabe des Mittelwerts
    cout << endl << "Mittelwert: " << mittel;

    return 0;
}

//Funktionsdefinition
//Erzeugt Zufallszahlen im Bereich von 1991-2018, 10-90
void erstelleArray(int *feld, int anzahl)
{
    srand(time(0)); //time(0) liefert die Sekunden seit 1.1.1970 --> dadurch
wird der Zufallsgenerator initialisiert

    for(int i=0; i<anzahl; i++)
    {
        feld[i]=rand() % 28 +1991;        //*(feld+i)=rand(0);
    }

}

//Ausgabe des Arrays
void ausgabe(int feld[], int anzahl)
{
    for(int j=0; j<anzahl; j++)
    {
        cout << feld[j] << endl;   //äquivalent zu *(feld+j);
    }
}
//Berechnet die Summe aller Zahlen im Array und gibt diese zurück
```

```c
int summe(int feld[], int anzahl) {

    int sum=0;

    //Berechne die Summe aller Zahlen des Arrays
    for (int i=0; i<anzahl; i++)
    {
        sum=sum+feld[i];
    }

    //Rückgabe des Inhalts von sum an das Hauptprogramm
    return sum;

}
//Berechnet den Mittelwert aller Zahlen und gibt diesen als Gleitkommazahl
zurück
float mittelwert(int summe, int anzahl)
{
    float mittel=0.0;
    mittel=summe/anzahl;

    //Rückgabe des Inhalts von mittel an das Hauptprogramm
    return mittel;
}
```

Ausgabe

```
Geben Sie bitte die Anzahl der Werte des Arrays an: 13
2017
2002
1996
1991
2000
2015
2005
1992
2001
1993
2006
1999
2003

Mittelwert: 2001
--------------------------------
Process exited after 3.698 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Sortieralgorithmen

Sortierproblem

Das Sortierproblem besteht darin, Datensätze eines gegebenen Datenbestands sortiert anzuordnen. Im Folgenden geht es darum, das Sortierproblem und seine Relevanz in der Informatik genauer zu betrachten.

Lösung des Sortierproblems

Zur Lösung des Sortierproblems sind eine Vielzahl an Verfahren entwickelt worden. Wir werden einige dieser Verfahren hier vorstellen und zur Verdeutlichung der Komplexitätsbetrachtungen in den folgenden Abschnitten nutzen. Um die Ideen und Komplexitätsbetrachtungen möglichst einfach zu gestalten, sollen nur Zahlen anstelle komplexer Datensätze betrachtet werden.

- Grundlagen Sortieralgorithmen
- Beschreibung verschiedener Sortieralgorithmen
- Sortieralgorithmen animiert
- http://www.sorting-algorithms.com/

Angabe

Sortiere ein vorgegebenes Feld (10,6,3,5,1) per SelectionSort in einer Funktion.

Lösung

```
/* Beispiel: Funktionen, Arrays und Sortieralgorithmen
   Filename: main.cpp
   Author: Lahmer
   Title: SelectionSort
   Description: Ein vorgegebenes Feld wird per SelectionSort (Suche nach
Minimum) sortiert
   Last Change:19.01.2018
*/
//Header-Dateien
#include <iostream>

//Namensraum
using namespace std;


//Funktionsdeklaration
void Sortieren(int feld[], int size);  //kein Rückgabewert, Übergabe: Feld,
Größe des Feldes
void ausgabe(int feld[], int size);        //kein Rückgabewert, Übergabe:
Feld, Größe des Feldes

int main(int argc, char** argv)
{

    int feld[5] = {10, 6, 3, 5, 1};

    //Funktionsaufruf
    Sortieren(feld, 5);                 //Aufruf der Funktion Sortieren
    ausgabe(feld, 5);                 //Aufruf der Funktion ausgabe

    return 0;
}


//Funktionsdefinition
void Sortieren(int feld[], int size){
    int position=0, hzahl;        //position=Position des Minimums,
hzahl=Hilfszahl für Zahlentausch
    for(int j=0; j<size; j++)      //Gehe von 0 bis 4 durch
    {
```

```
        position=j;    //1. Position des sortierten Teiles des Arrays (1. Durchlauf 0, 2.Durchlauf = 1; 3. Durchlauf =2; ... =4)
        for(int i=j+1; i<size; i++)   //Suche Minimum - beginne immer bei j+1 ; da ein Vergleich von feld[i]<feld[position] keinen Sinn macht, da sie im ersten Schritt auf dieselbe Zahl zeigen
        {
            if(feld[i]<feld[position])    //Schaue ob Feld[i] kleiner als das aktuelle Minimum ist
            {
                position=i;                //Wenn ja, speichere die aktuelle Position i in position
            }
        }
        hzahl=feld[j];                //speichere die Zahl an Position j in die Hilfsvariable hzahl
        feld[j] = feld[position];    //speichere das Minimum an der Position position an die Position j
        feld[position] = hzahl;        //speichere hzahl an die Position position
    }
}

//Funktionsdefinition
void ausgabe(int feld[], int size)   //Ausgabe des Feldes
{

    for(int i=0; i<size;i++)
    {
        cout << feld[i] << endl;
    }

}
```

Ausgabe

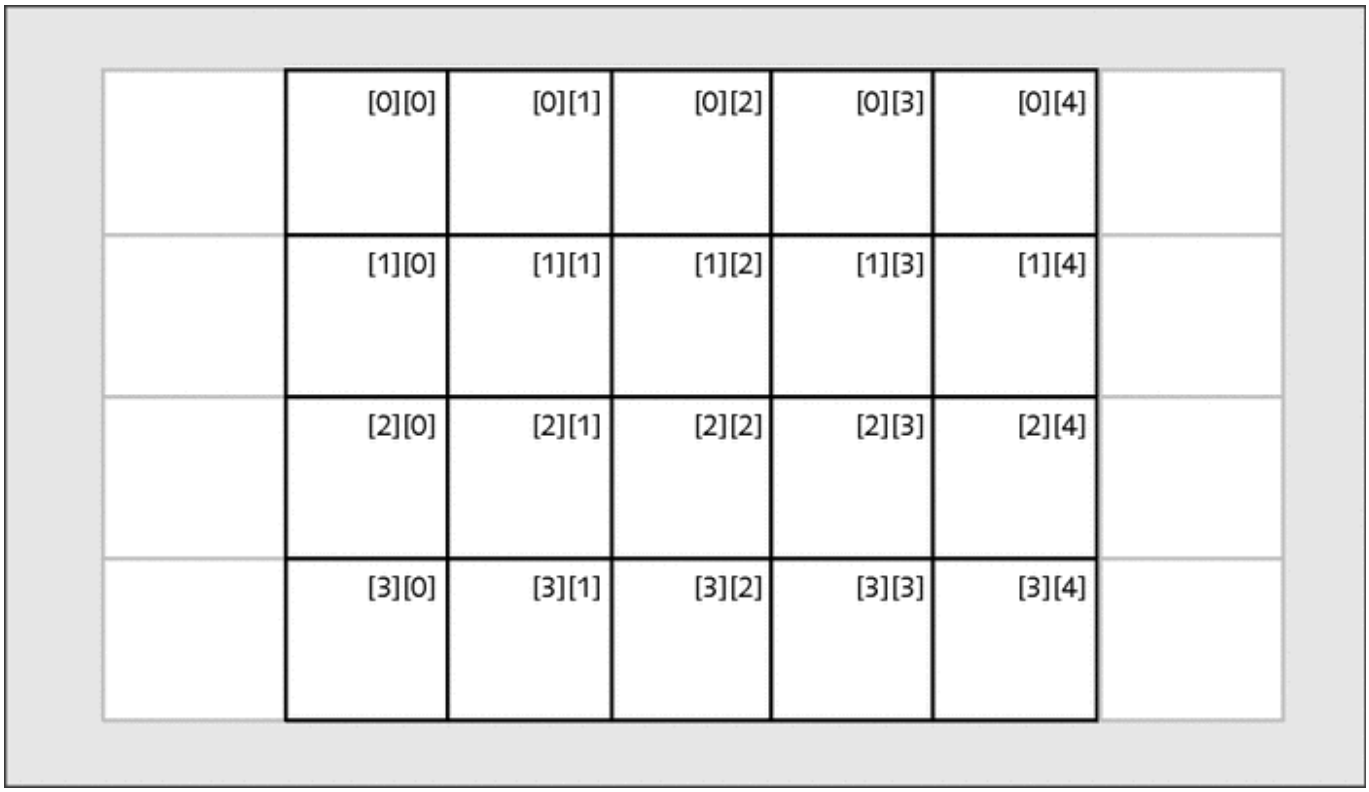
```
1
3
5
6
10

--------------------------------
Process exited after 0.08686 seconds with return value 0
Drücken Sie eine beliebige Taste . . .
```

Mehrdimensionale Arrays

Arrays, wie sie bisher besprochen wurden, können Sie sich als einen Strang von hintereinander aufgereihten Zahlen vorstellen. Man spricht dann von eindimensionalen Arrays oder Feldern. Es ist aber auch möglich, Arrays mit mehr als nur einer Dimension zu verwenden:

```
int Matrix[4][5];    /* Zweidimensional - 4 Zeilen x 5 Spalten */
```

Hier wurde z. B. ein zweidimensionales Array mit dem Namen Matrix definiert. Dies entspricht im Prinzip einem Array, dessen Elemente wieder Arrays sind. Sie können sich dieses Feld wie bei einer Tabellenkalkulation vorstellen.



```
/* Beispiel: Mehrdimensionale Arrays
   Filename: main.cpp
   Author: Lahmer
   Title: Schiffe versenken
   Description: Es soll eine vereinfachte Variation von Schiffe versenken
erstellt werden.
   Last Change: 19.01.2018
*/

//Header-Dateien
#include <iostream>

//Namespace
using namespace std;

//Globale Variablen
#define SPALTEN 5
#define ZEILEN 5
#define SCHIFFLAENGE 3


//Funktionsprototyp
void initializeArray(char feld[ZEILEN][SPALTEN]);  //Rückgabewert void,
Übergabe per Zeiger
void printArray(char feld[ZEILEN][SPALTEN]);   //Rückgabewert void, Übergabe
per Zeiger
int check(char feld[ZEILEN][SPALTEN], char schiff[ZEILEN][SPALTEN], int
zeile, int spalte);  //Rückgabewert int, Übergabe per Zeiger, per Zeiger,
per Wert & per Wert


//Hauptprogramm
int main(int argc, char** argv) {

    //Lokale Variablendeklaration
    //Mehrdimensionales Character-Feld für die Anzeige
    char feld[ZEILEN][SPALTEN];
    //Mehrdimensionales Character-Feld für das Schiff
    char schiff[ZEILEN][SPALTEN];
    int zeile=0, spalte=0, treffer=0;


    //Feld für das Spielfeld mit Werten initialisieren
    initializeArray(feld);

    //Feld für das Schiff  mit Werten initialisieren
    initializeArray(schiff);

    //Schiff verstecken
    schiff[1][2]='X';
```

```
    schiff[1][3]='X';
    schiff[1][4]='X';

    printArray(feld);

    //Do-While Schleife solange nicht das ganze Schiff versenkt ist
    do {

        cout << endl << "Geben Sie die Zeilennummer ein:";
        cin >> zeile;
        cout << endl << "Geben Sie die Spaltennummer ein:";
        cin >> spalte;

        //Die Treffer mitzählen, um zu wissen wann das Schiff versenkt ist
        treffer=treffer+check(feld,schiff,zeile,spalte);

        //Spielfeld ausgeben
        printArray(feld);

    }while(treffer<SCHIFFLAENGE);

    cout << endl << endl << "!!!!!!!!!!! HERZLICHEN GLUECKWUNSCH SIE HABEN
DAS SCHIFF VERSENKT !!!!!!!!!!!!!!!";

    return 0;
}

//Alle Feldelemente mit # initialisieren
void initializeArray(char feld[ZEILEN][SPALTEN]) {

    for (int i=0; i<ZEILEN;i++)
    {
        for (int j=0; j<SPALTEN;j++)
        {
            feld[i][j]='#';
        }
    }
}

//Spielfeld ausgeben
void printArray(char feld[ZEILEN][SPALTEN]) {

    cout << endl << endl << "-----------------------------------------------
--------------------";

    cout << endl << "Spielfeld: " << endl;
    cout << "   ";
    for (int i=0; i<ZEILEN;i++)
    {
        cout << i << " ";
    }
```

```
    cout << endl;
    for (int i=0; i<ZEILEN;i++)
    {
        cout << i << " ";
        for (int j=0; j<SPALTEN;j++)
        {
            cout << feld[i][j] << " ";
        }
        cout << endl;
    }
    cout  << "-----------------------------------------------------------------
-----" << endl << endl;

}


//Prüfen, ob ein Schiff getroffen wurde
int check(char feld[ZEILEN][SPALTEN], char schiff[ZEILEN][SPALTEN], int
zeile, int spalte) {

    if(schiff[zeile][spalte]=='X')
    {
        cout << endl << "!!!! TREFFER !!!!!" << endl;
        feld[zeile][spalte]='X';

        return 1;
    }
    else {
        cout << endl << "!!!! KEIN TREFFER !!!!!" << endl;
        feld[zeile][spalte]='-';

        return 0;
    }

    return 0;

}
```

```
/* Beispiel: Schiffe versenken (Strukturen, Schleifen, Funktionen,
Zufallszahlen)
   Filename: main.cpp
   Author: Lahmer
   Title: Schiffe versenken
   Description: Es soll eine weitere Variation vom Spiel Schiffe versenken
erstellt werden, wo die Schiffe automatisch gesetzt werden.
   Last Change: 30.01.2018
*/


#include <iostream>
#include <time.h>       //Bibliothek für time(NULL)
#include <stdlib.h>     //Bibliothek für rand(), srand()
#include <windows.h>
#include <string>
#include <conio.h>
#include <fstream>

using namespace std;

//GLOBALE VARIABLEN
#define ZEILEN 10
#define SPALTEN 10
#define LAENGE 3
#define RICHTUNGEN 4
#define ANZSCHIFFE 5

void intializeArray(char array[ZEILEN][SPALTEN]);  //initialisiere Array-
Elemente mit #
void printArray(char array[ZEILEN][SPALTEN]);      //gib Array aus
int check(char Schiff[ZEILEN][SPALTEN], char Feld[ZEILEN][SPALTEN], int y,
int x); //Prüfe Array ob Schiff getroffen wurde
bool setzeSchiff(char Schiff[ZEILEN][SPALTEN], struct SchiffInfo
schiff1);//Setze Schiff zufällig

struct SchiffInfo{
    int laenge;         //Länge von 2-5
    int startX;         //Anfangsposition auf der X-Achse
    int startY;         //Anfangsposition auf der Y-Achse
    int richtung;    //0->oben, 1->rechts oben, 2->rechts, 3->rechts
unten,4->unten, 5->links unten, 6->links, 7-> links oben
};



int main(int argc, char** argv) {

    SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),15);
```

```
    //LOKALE VARIABLEN
    char feld[ZEILEN][SPALTEN];        //Deklaration eines mehrdimensionalen
Arrays
    char schiff[ZEILEN][SPALTEN];    //Deklaration eines mehrdimensionalen
Arrays
    int zeile=0, spalte=0, zaehler=0, versuche=0;
    int schiffselemente=0;
    bool Schiffversteckt=false;
    int schiffanz=ANZSCHIFFE;
    string name;

    do{

        cout << "Geben Sie Ihren Namen ein: ";
        cin >> name;

        cout << endl << "Geben Sie die Anzahl der zu platzierenden Schiffe
ein: ";
        cin >> schiffanz;

        system("cls");


        intializeArray(feld);        //Array namens feld initialisieren
        intializeArray(schiff);        //Array namens schiff initialisieren

        struct SchiffInfo s1;
        srand(time(NULL));

        do {
            s1.laenge=rand()%4+2;        //Bestimmen der Schiffslänge
            s1.startX=rand()%SPALTEN;    //Bestimmen der Startposition auf
der X-Achse
            s1.startY=rand()%ZEILEN;    //Bestimmen der Startposition auf
der Y-Achse
            s1.richtung=rand()%RICHTUNGEN;    //Bestimmen der Setzrichtung
des Schiffes

            cout << "Laenge: "<< s1.laenge << endl;
            cout << "StartY: "<< s1.startY << endl;
            cout << "StartX: "<< s1.startX << endl;
            cout << "Richtung: "<< s1.richtung << endl;

            //Schiff verstecken
            Schiffversteckt=setzeSchiff(schiff,s1);

        /* Anzahl Schiffe */
            if(Schiffversteckt)
            {
                schiffanz--;
```

```
                schiffselemente=schiffselemente+s1.laenge;
            }

        }while(schiffanz>0);        //Solange Anzahl der Schiffe > 0

        printArray(feld);        //Array feld ausgeben
        printArray(schiff);        //Array schiff ausgeben

        do{

            cout << endl << "Geben Sie bitte die Zeilennummer ein:";
            cin >> zeile;
            cout << endl << "Geben Sie bitte die Spaltennummer ein:";
            cin >> spalte;

            system("cls");        //Löscht die Konsolenausgabe

            zaehler=zaehler+check(schiff, feld, zeile, spalte);
//Prüfen ob Schiff getroffen wurde
            versuche++;        //Zählen der Versuche

            printArray(feld);        //Array namens feld ausgeben

        }while(zaehler<schiffselemente);

        cout << endl<< "Sie haben das Schiff/die Schiffe nach insgesamt " <<
versuche << " Versuch(en) versenkt!! GRATULATION!!" << endl;
        cout << endl << "Sie haben eine Trefferwahrscheinlichkeit von " <<
float(schiffselemente)/float(versuche) << endl;

        ofstream fileout;
        fileout.open("highscore.txt");
        fileout << name << ": " << float(schiffselemente)/float(versuche);
        fileout.close();

        cout << endl << "Wollen Sie das Spiel nochmals spielen (j/n)?";
    }while(getch()=='j');

    return 0;
}

//Initialisiert das übergebene Array mit #
void intializeArray(char array[ZEILEN][SPALTEN])
{
    for(int i=0;i<ZEILEN;i++)
    {
        for(int j=0;j<SPALTEN;j++)
        {
            array[i][j]='#';
        }
    }
```

```
}
//Gibt das übergebene Array aus
void printArray(char array[ZEILEN][SPALTEN])
{

    cout << endl << "Spielfeld: ";
    cout << endl << endl;

    cout << "  ";
    for(int i=0;i<SPALTEN;i++)
    {
        cout << i << " ";
    }
    cout << endl;
    for(int i=0;i<ZEILEN;i++)
    {
        cout << i << " ";
        for(int j=0;j<SPALTEN;j++)
        {
            if(array[i][j]=='X')
            {
                SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),4);
            }
            cout << array[i][j] << " ";
            SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE),15);
        }
        cout << endl;
    }
}
//Prüft ob Schiff getroffen wurde (1) oder nicht (0)
int check(char Schiff[ZEILEN][SPALTEN], char Feld[ZEILEN][SPALTEN], int y,
int x)
{
    if(Schiff[y][x]=='X')
    {
        cout << endl << "!!!! TREFFER !!!!!" << endl;
        Feld[y][x]='X';

        return 1;
    }
    else {
        cout << endl << "!!!! KEIN TREFFER !!!!!" << endl;
        Feld[y][x]=' ';

        return 0;
    }
}

//Setze das übergebene schiff1 im Array namens Schiff, falls möglich
```

```
bool setzeSchiff(char Schiff[ZEILEN][SPALTEN], struct SchiffInfo schiff1)
{
    int zaehler=0;
    if(Schiff[schiff1.startY][schiff1.startX]!='X')
    {
        zaehler=1;    //Für jedes Schiffteil, welches platziert wird -> zähler++
        //Prüfen der Richtung 0 = nach links oben  (Y wird weniger, X wird weniger)
        if(schiff1.richtung==0)
        {
            if(schiff1.startY-(schiff1.laenge-1)>=0 && schiff1.startX-(schiff1.laenge-1)>=0)        //Prüft ob in Y- und X-Richtung genung Platz für das Schiff ist
            {
                //Schiff hat genug Platz
                for(int i=1;i<schiff1.laenge;i++)        //Prüft jede Koordinate ob bereits ein Schiff vorhanden ist
                {
                    if(Schiff[schiff1.startY-i][schiff1.startX-i]!='X')
                    {
                        //cout << "Position ist frei";
                        zaehler++;
                    }
                }
            }

            if(zaehler==schiff1.laenge)
            {
                cout << "Schiff wird platziert - Zaehler: " << zaehler << endl;
                //Schiff wird erst jetzt platziert!!!
                for(int i=0;i<schiff1.laenge;i++)
                {
                    Schiff[schiff1.startY-i][schiff1.startX-i]='X';

                }
                return true;

            }
            else
            {
                return false;
            }
        }

        //Prüfen der Richtung 1 == nach oben (Y wird weniger, X bleibt gleich)
        zaehler=1;
        if(schiff1.richtung==1)
        {
```

```
            if(schiff1.startY-(schiff1.laenge-1)>=0)
            {
                for(int i=schiff1.startY-1;i>schiff1.startY-
schiff1.laenge;i--)
                {
                    if(Schiff[i][schiff1.startX]!='X')
                    {
                        zaehler++;
                    }
                }
            }

            if(zaehler==schiff1.laenge)
            {
                cout << "Schiff wird platziert - Zaehler: " << zaehler <<
endl;
                //Schiff wird erst jetzt platziert!!!
                for(int i=0;i<schiff1.laenge;i++)
                {
                    Schiff[schiff1.startY-i][schiff1.startX]='X';

                }
                return true;

            }
            else
            {
                return false;
            }
        }


        //Prüfen der Richtung 2 == nach oben rechts (Y wird weniger, X wird
mehr)
        zaehler=1;
        if(schiff1.richtung==2)
        {
            if(schiff1.startY-(schiff1.laenge-1)>=0 &&
schiff1.startX+(schiff1.laenge-1)<SPALTEN)
            {
                for(int i=1;i<schiff1.laenge;i++)        //Prüft jede
Koordinate ob bereits ein Schiff vorhanden ist
                {
                    if(Schiff[schiff1.startY-i][schiff1.startX+i]!='X')
                    {
                        //cout << "Position ist frei";
                        zaehler++;
                    }
                }
            }
```

```
            if(zaehler==schiff1.laenge)
            {
                cout << "Schiff wird platziert - Zaehler: " << zaehler <<
endl;
                //Schiff wird erst jetzt platziert!!!
                for(int i=0;i<schiff1.laenge;i++)
                {
                    Schiff[schiff1.startY-i][schiff1.startX+i]='X';

                }
                return true;

            }
            else
            {
                return false;
            }

        }

        //Prüfen der Richtung 3 == nach rechts (Y bleibt gleich, X wird
mehr)
        zaehler=1;
        if(schiff1.richtung==3)
        {
            if(schiff1.startX+(schiff1.laenge-1)<SPALTEN)
            {
                for(int i=1;i<schiff1.laenge;i++)       //Prüft jede
Koordinate ob bereits ein Schiff vorhanden ist
                {
                    if(Schiff[schiff1.startY][schiff1.startX+i]!='X')
                    {
                        //cout << "Position ist frei";
                        zaehler++;
                    }
                }
            }

            if(zaehler==schiff1.laenge)
            {
                cout << "Schiff wird platziert - Zaehler: " << zaehler <<
endl;
                //Schiff wird erst jetzt platziert!!!
                for(int i=0;i<schiff1.laenge;i++)
                {
                    Schiff[schiff1.startY][schiff1.startX+i]='X';

                }
                return true;
```

```
            }
            else
            {
                return false;
            }

        }
    }
    return false;

}
```

Struktur - eine zusammengesetzter Datentyp

Mit Arrays können Variablen gleichen Typs zusammengestellt werden. In der realen Welt gehören aber meist Daten unterschiedlichen Typs zusammen. So hat ein Auto einen Markennamen und eine Typbezeichnung, die als Zeichenkette unterzubringen ist. Dagegen eignet sich für Kilometerzahl und Leistung eher der Typ Integer. Für den Preis bietet sich der Typ float an. Bei bestimmten Autohändlern könnte auch double erforderlich sein. Alles zusammen beschreibt ein Auto.

Modell

Vielleicht werden Sie einwerfen, dass ein Auto noch mehr Bestandteile hat. Da gibt es Bremsscheiben, Turbolader und Scheibenwischer. Das ist in der Realität richtig. Ein Programm interessiert sich aber immer nur für bestimmte Eigenschaften, die der Programmierer mit dem Kunden zusammen festlegt. Unser Beispiel würde für einen kleinen Autohändler vielleicht schon reichen. Eine Autovermietung interessiert sich vielleicht überhaupt nicht für den Wert des Autos, aber möchte festhalten, ob es für Nichtraucher reserviert ist. Eine Werkstatt dagegen könnte sich tatsächlich für alle Teile interessieren. Ein Programm, das die Verteilung der Firmenfahrzeuge verwaltet, interessiert sich vielleicht nur für das Kennzeichen. Es entsteht also ein Modell eines Autos, das bestimmte Bestandteile enthält und andere vernachlässigt, je nachdem was das Programm benötigt. Bereits in C gab es für solche Zwecke die Struktur, die mehrere Variablen zu einer zusammenfasst. Das Schlüsselwort für die Bezeichnung solch zusammengesetzter Variablen lautet struct. Nach diesem Schlüsselwort folgt der Name des neuen Typen. In dem folgenden geschweiften Klammernblock werden die Bestandteile der neuen Struktur aufgezählt. Diese unterscheiden sich nicht von der bekannten Variablendefinition. Den Abschluss bildet ein Semikolon.

struct

Um ein Auto zu modellieren, wird ein neuer Variablentyp namens TAutoTyp geschaffen, der ein Verbund mehrerer Elemente ist.

```
struct TAutoTyp // Definiere den Typ
{
    char Marke[MaxMarke];
    char Modell[MaxModell];
    long km;
    int kW;
    float Preis;
};  // Hier vergisst man leicht das Semikolon!
```

Syntaxbeschreibung

Das Schlüsselwort struct leitet die Typdefinition ein. Es folgt der Name des neu geschaffenen Typs, hier TAutoTyp. In dem nachfolgenden geschweiften Klammerpaar werden alle Bestandteile der

Struktur nacheinander aufgeführt. Am Ende steht ein Semikolon, das man selbst als erfahrener Programmierer immer wieder einmal vergisst. Variablendefinition Damit haben wir den Datentyp TAutoTyp geschaffen. Er kann in vieler Hinsicht verwendet werden wie der Datentyp int. Sie können beispielsweise eine Variable von diesem Datentyp anlegen. Ja, Sie können sogar ein Array und einen Zeiger von diesem Datentyp definieren.

```
TAutoTyp MeinRostSammler; // Variable anlegen
TAutoTyp Fuhrpark[100];   // Array von Autos
TAutoTyp *ParkhausKarte;  // Zeiger auf ein Auto
```

Elementzugriff

Die Variable MeinRostSammler enthält nun alle Informationen, die in der Deklaration von TAutoTyp festgelegt sind. Um von der Variablen auf die Einzelteile zu kommen, wird an den Variablenname ein Punkt und daran der Name des Bestandteils gehängt.

```
// Auf die Details zugreifen
MeinRostSammler.km = 128000;
MeinRostSammler.kW = 25;
MeinRostSammler.Preis = 25000.00;
```

Zeigerzeichen

Wenn Sie über einen Zeiger auf ein Strukturelement zugreifen wollten, müssten Sie über den Stern referenzieren und dann über den Punkt auf das Element zugreifen. Da aber der Punkt vor dem Stern ausgewertet wird, müssen Sie eine Klammer um den Stern und den Zeigernamen legen.

```
TAutoTyp *ParkhausKarte = 0;      // Erst einmal keine Zuordnung
ParkhausKarte = &MeinRostSammler; // Nun zeigt sie auf ein Auto
(*ParkhausKarte).Preis = 12500;   // Preis für MeinRostSammler
```

Das mag zwar logisch sein, aber es ist weder elegant noch leicht zu merken. Zum Glück gibt es in C und C++ eine etwas hübschere Variante, über einen Zeiger auf Strukturelemente zuzugreifen. Dazu wird aus Minuszeichen und Größer-Zeichen ein Symbol zusammengesetzt, das an einen Pfeil erinnert.

```
ParkhausKarte->Preis = 12500;
L-Value
Strukturen sind L-Values. Sie können also auf der linken Seite einer
Zuweisung stehen. Andere Strukturen des gleichen Typs können ihnen
zugewiesen werden. Dabei wird die Quellvariable Bit für Bit der Zielvariable
zugewiesen.
TAutoTyp MeinNaechstesAuto, MeinTraumAuto;
MeinNaechstesAuto = MeinTraumAuto;
```

Trotzdem die beiden Strukturvariablen nach dieser Operation ganz offensichtlich gleich sind, kann man dies nicht einfach durch eine Anwendung des doppelten Gleichheitszeichen nachprüfen. Sie

können bei Strukturen die Typdeklaration und die Variablendefinition zusammenfassen, indem der Name der Variablen direkt nach der geschweiften Klammer eingetragen wird.

```
struct // hier wird kein Typ namentlich festgelegt
{
    char Marke[MaxMarke];
    char Modell[MaxModell];
    long km;
    int kW;
    float Preis;
} MeinErstesAuto, MeinTraumAuto;
```

Hier werden im Beispiel die Variablen MeinErstesAuto und MeinTraumAuto gleich mit ihrer Struktur definiert. Werden auf diese Weise gleich Variablen dieser Struktur gebildet, muss ein Name für den Typ nicht unbedingt angegeben werden. Damit ist dann natürlich keine spätere Erzeugung von Variablen dieses Typs möglich. Initialisierung Auch Strukturen lassen sich initialisieren. Dazu werden wie bei den Arrays geschweifte Klammern verwendet. Auch hier werden die Werte durch Kommata getrennt.

```
TAutoTyp JB = {"Aston Martin", "DB5", 12000, 90, 12.95};
TAutoTyp GWB = {0};
```

Objektorientierte Programmierung (OOP)

Am Anfang steht wohl die Frage was Objektorientierte Programmierung überhaupt ist. Dies ist nicht ganz einfach zu erklären und es gibt viele Definitionen die dies versuchen. Grob gesagt ist Objektorientierte Programmierung (OOP) ein Verfahren zur Strukturierung von Programmen, bei dem Programmlogik zusammen mit Zustandsinformationen (Datenstrukturen) in Einheiten, den Objekten, zusammengefasst werden. Es ist also möglich mehrere Objekte des gleichen Typs zu haben. Jedes dieser Objekte hat dann seinen individuellen Zustand. Dies bietet die Möglichkeit einer besseren Modularisierung der Programme, sowie einer höheren Wartbarkeit des Quellcodes.

Klassen

Die Klasse (class) ist die zentrale Datenstruktur in C + +. Sie kapselt zusammengehörige Daten und Funktionen vom Rest des Programmes ab. Sie ist das Herz der objektorientierten Programmierung (OOP).

Klassen sind ein erweitertes Konzept von Datenstrukturen denen es erlaubt ist, außer Daten, noch Methoden zu beinhalten. Ein Objekt ist eine Instanz einer Klasse, welches sich die Eigenschaften der Klasse zunutze machen kann. Es ist möglich mehrere Objekte einer Klasse zu instanziieren, wobei jedes dieser Objekte zwar die selben Eigenschaften hat, intern aber einen ganz individuellen Zustand haben kann.

Beispiel: Zwei Instanzen der Klasse „Gegner“ werden erzeugt. Nachdem ein Gegner Schaden erlitten hat sind seine Lebenspunkte auf 50 gesunken. Die des anderen Gegners haben allerdings noch den Wert 100.

Klassen werden normalerweise mit dem Schlüsselwort **class** deklariert. Außerdem haben sie immer folgendes Format:

```
class Klassenname
{
Zugriffsspezifizierung 1:
Member 1;
...
Zugriffsspezifizierung 2:
Member 2;
...
...
};
```

Der Klassenname identifiziert die Klasse, der Objektname ist eine optionale Liste von Namen von Objekten dieser Klasse. Der Körper der Klasse wird durch geschweifte Klammern gekennzeichnet und kann verschiedene Member (Methoden, Membervariablen) beinhalten. Diesen können verschiedene Zugriffsspezifierungen zugewiesen werden.

Zugriffsrechte

public: Auf Members kann von überall zugegriffen werden von wo das Objekt sichtbar ist

private: Auf Member kann nur innerhalb anderer Member zugegriffen werden oder aus befreundeten Klassen und Funktionen

protected: Members sind verfügbar für Member der selben Klasse, befreundeten Funktionen/Klassen und Abgeleiteten Klassen

Attribute

Die Festlegung, welche Daten zu einem Typ gehören, erfolgt bei der Definition der Klasse. Die Objekte enthalten jeweils unabhängig voneinander einen Satz von Datenkomponenten (Attributen). Ihre Startwerte können durch Konstruktoren festgelegt werden. C + + erlaubt auch die Zuweisung von Anfangswerten bei der Definition:

```
class Enemy {
    public:   //Zugriffsrecht public
        void setHealth(int h);
        int getHealth();
        string name;
                int id;
    private:  //Zugriffsrecht private
        int health;
};
```

Elementzugriff

Im Beispiel davor wird ein Objekt vom **Typ Enemy** angelegt. Das **Objekt** enthält die **3 Variablen health (integer), id (int) und name (String)**, wie es in der Klassendefinition zu sehen ist. Um auf eine öffentliche Elementvariable zugreifen zu können, wird an den Objektnamen ein **Punkt** und dann der in der Klassendefinition verwendete Elementname gehängt. Im Beispiel wird der Name auf Andreas gesetzt.

```
    ...
    //Zugriff auf öffentliche Variable name
    e1.name="Andreas";
    //Zugriff auf öffentliche Variable id
    e1.id=1;
    ...
    //e1.health=100; ist nicht erlaubt, da health das Schlüsselwort private
besitzt!!!
```

Klassenmethoden

Nun können Sie ein Objekt von der Klasse Enemy erzeugen. Aber was nützt Ihnen die schönste Datenstruktur in Ihrem Programm, wenn sie nicht durch Funktionen zum Leben erweckt wird? Sie werden z.B. einen Namen und Lebenspunkte eingeben und ausgeben wollen. Vielleicht wollen Sie die Lebenspunkte verringern oder erhöhen. Kurz gesagt, ein Datenverbund ist nichts wert ohne Funktionen, die auf ihn wirken. Aber die Funktionen sind auch nur im Zusammenhang mit ihrem Datenverbund sinnvoll. Aus diesem Grund werden die Funktionen ebenso in die Klasse integriert wie die Datenelemente. Eine Funktion, die zu einer Klasse gehört, nennt man Elementfunktion oder auf englisch member function. In anderen objektorientierten Programmiersprachen spricht man auch von einer Methode oder Operation. Aufruf So, wie Sie auf Datenelemente nur über ein real existierendes Objekt, also eine Variable dieser Klasse zugreifen können, kann auch eine Funktion nur über ein Objekt aufgerufen werden. Objekt und Funktionsnamen werden dabei durch einen Punkt getrennt. Die Funktion arbeitet mit den Daten des Objekts, über das sie gerufen wurde. Aus prozeduraler Sicht könnte man es so sehen, dass eine Elementfunktion immer bereits einen Parameter mit sich trägt, nämlich das Objekt, über das sie aufgerufen wurde.

Beispiel:

```
#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

//Klasse Enemy
class Enemy {
    public:   //Zugriffsrecht public
        void setHealth(int h);
        int getHealth();
        string name;
                int id;
    private:  //Zugriffsrecht private
        int health;
};
//Methode setHealth zum Setzen der Lebenspunkte
void Enemy::setHealth(int h)
{
    health=h;
}

//Methode getHealth() zum Auslesen der Lebenspunkte
int Enemy::getHealth()
{
    return health;
}

//Hauptprogramm
int main()
{
    Enemy e1;   //Objekt e1 der Klasse Enemy wird erzeugt

    //Zugriff auf öffentliche Variable name
```

```
    e1.name="Andreas";
    //Zugriff auf öffentliche Variable id
    e1.id=1;

    //Aufruf der Methode setHealth
    e1.setHealth( 100 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";
    //Aufruf der Methode setHealth
    e1.setHealth( 50 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.n";

    return 0;
}
```

Methodendefinition innerhalb einer Klasse

Alternativ können Sie die Elementfunktion auch direkt in der Klasse definieren. Das wird leicht unübersichtlich, darum sollten Sie das nur bei sehr kurzen Funktionen tun.

```
#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

class Enemy {
    public:     //Zugriffsrecht public
                /***** INLINE - METHODENDEFINITION *****/
        //Methode setHealth zum Setzen der Lebenspunkte
        void setHealth(int h)
        {
            health=h;
        }
        //Methode getHealth() zum Auslesen der Lebenspunkte
        int getHealth()
        {
            return health;
        }

        string name;
                int id;


    private:     //Zugriffsrecht private
```

```
        int health;
};

//Hauptprogramm
int main()
{
    Enemy e1;   //Objekt e1 der Klasse Enemy wird erzeugt

    //Zugriff auf öffentliche Variable name
    e1.name="Andreas"

    //Aufruf der Methode setHealth
    e1.setHealth( 100 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";
    //Aufruf der Methode setHealth
    e1.setHealth( 50 );
    //Aufruf der Methode getHealth und Zugriff auf die öffentliche Variable
name
    cout << "Der Gegner " << e1.name << " hat " << e1.getHealth() << "
Lebenspunkte.\n";

    return 0;
}
```

Aufruf

So, wie Sie auf Datenelemente nur über ein real existierendes Objekt, also eine Variable dieser Klasse zugreifen können, kann auch eine Funktion **nur über ein Objekt aufgerufen werden**.
Objekt und **Funktionsnamen** werden dabei durch einen **Punkt** getrennt. Die Funktion arbeitet mit den Daten des Objekts, über das sie gerufen wurde. Aus prozeduraler Sicht könnte man es so sehen, dass eine Elementfunktion immer bereits einen Parameter mit sich trägt, nämlich das Objekt, über das sie aufgerufen wurde.

```
    ....
    Enemy e1;   //Objekt e1 der Klasse Enemy wird erzeugt

    //Aufruf der Methode setHealth
    e1.setHealth( 100 );
    ....
```

Konstruktor

Die Elementfunktion, die beim Erzeugen eines Objekts aufgerufen wird, nennt man Konstruktor. In dieser Funktion können Sie dafür sorgen, dass alle Elemente des Objekts korrekt initialisiert sind. Der Konstruktor trägt immer den Namen der Klasse selbst und hat keinen Rückgabetyp, auch nicht void.

Der Standardkonstruktor hat keine Parameter.

```
class Enemy {
    public:     //Zugriffsrecht public

                /**** Konstruktor ****/
                Enemy()
                {
                       health=0;
                       name="";
                       id=0;
                }

        string name;
                int id;


    private:     //Zugriffsrecht private
        int health;
};
```

Dekonstruktor

Im Falle einer Datumsklasse wäre es sinnvoll, dass der Konstruktor alle Elemente auf 0 setzt. Daran kann jede Elementfunktion leicht erkennen, dass das Datum noch nicht festgelegt wurde. Sie könnten alternativ das aktuelle Datum ermitteln und eintragen. Im Beispiel ist auch ein Destruktor definiert worden, obwohl er im Falle eines Datums keine Aufgabe hat.

```
class Enemy {
    public:     //Zugriffsrecht public

                /**** Destruktor ****/
                ~Enemy()
                {
                     cout << "Ressourcen von " <<  name << " wurden
freigegeben!";
                }

        char name[50];
                int id;


    private:     //Zugriffsrecht private
        int health;
};
```

Überladen von Konstruktor

Konstruktoren können genauso überladen werden wie normale Funktionen auch. Es kann neben dem Standardkonstruktor auch mehrere weitere Konstruktoren mit verschiedenen Parametern geben. Der Compiler wird anhand der Aufrufparameter unterscheiden, welcher Konstruktor verwendet wird.

```
class Enemy {
    public:     //Zugriffsrecht public


                /**** Konstruktor ****/
                Enemy()
                {
                     name="Max Mustermann";
                }
                Enemy(string n)
                {
                     name=n;
                }
                Enemy(string n, int h)
                {
                     name=n;
                     health=h;
                }
                /**** Destruktor ****/
                ~Enemy()
                {
                     cout << "Ressourcen von " <<  name << " wurden
freigegeben!";
                }

        char name[50];
                int id;


    private:     //Zugriffsrecht private
        int health;
};
```

Je nachdem wie viele Parameter beim Konstruktoraufruf übergeben werden, wird der passende Konstruktor ausgewählt. Existiert gar kein Konstruktor so wird vom Compiler ein leerer Konstruktor eingefügt.

```
int main (void)
{

     Enemy e;                    //führt den Standardkonstruktor Enemy() aus
-> name="Max Mustermann"
     Enemy e("Fritz Phantom");   //führt den 2. Konstruktor mit den
passenden Argumenten aus -> name="Fritz Phantom"
     Enemy e("Hans Wurst", 100); //führt den 3. Konstruktor mit den
passenden Argumenten aus -> name="Hans Wurst", health=100
```

```
    return 0;
}
```

Vererbung

Die Informatik steckt voller schöner Analogien. So hat die objektorientierte Programmierung den Begriff der »Vererbung« eingeführt, wenn eine Klasse von einer anderen Klasse abgeleitet wird und deren Eigenschaften übernimmt.

Basisklasse

Eine Klasse kann als Basis zur Entwicklung einer neuen Klasse dienen, ohne dass ihr Code geändert werden muss. Dazu wird die neue Klasse definiert und dabei angegeben, dass sie eine abgeleitete Klasse der Basisklasse ist. Daraufhin gehören alle öffentlichen Elemente der Basisklasse auch zur neuen Klasse, ohne dass sie erneut deklariert werden müssen. Man sagt, die neue Klasse erbt die Eigenschaften der Basisklasse.

Spezialisierung

Durch die Elemente, die in der abgeleiteten Klasse definiert werden, wird die abgeleitete Klasse zu einem besonderen Fall der Basisklasse. Sie besitzt alle Eigenschaften der Basisklasse. Sie können neue Elemente hinzufügen. Am folgenden Beispiel wird deutlich, warum das Hinzufügen von Eigenschaften eine Spezialisierung ist.

Ein Beispiel

Aus Sicht eines Computerprogramms haben alle **Personen** (=Basisklasse)

- Namen
- Adressen und
- Telefonnummern.

Geschäftspartner haben darüber hinaus eine

- Bankverbindung.

Da die Geschäftspartner auch Personen sind, haben sie neben ihrer Bankverbindung auch Namen, Adressen und Telefonnummern (von der Basisklasse).

Einige Geschäftspartner können auch **Kunden** sein. **Kunden** haben zusätzlich zu den Eigenschaften eines Geschäftspartners noch eine

- Lieferanschrift.

Lieferanten sind keine Kunden, aber auch Geschäftspartner. Sie haben noch

- eine Anzahl an offenen Rechnungen.

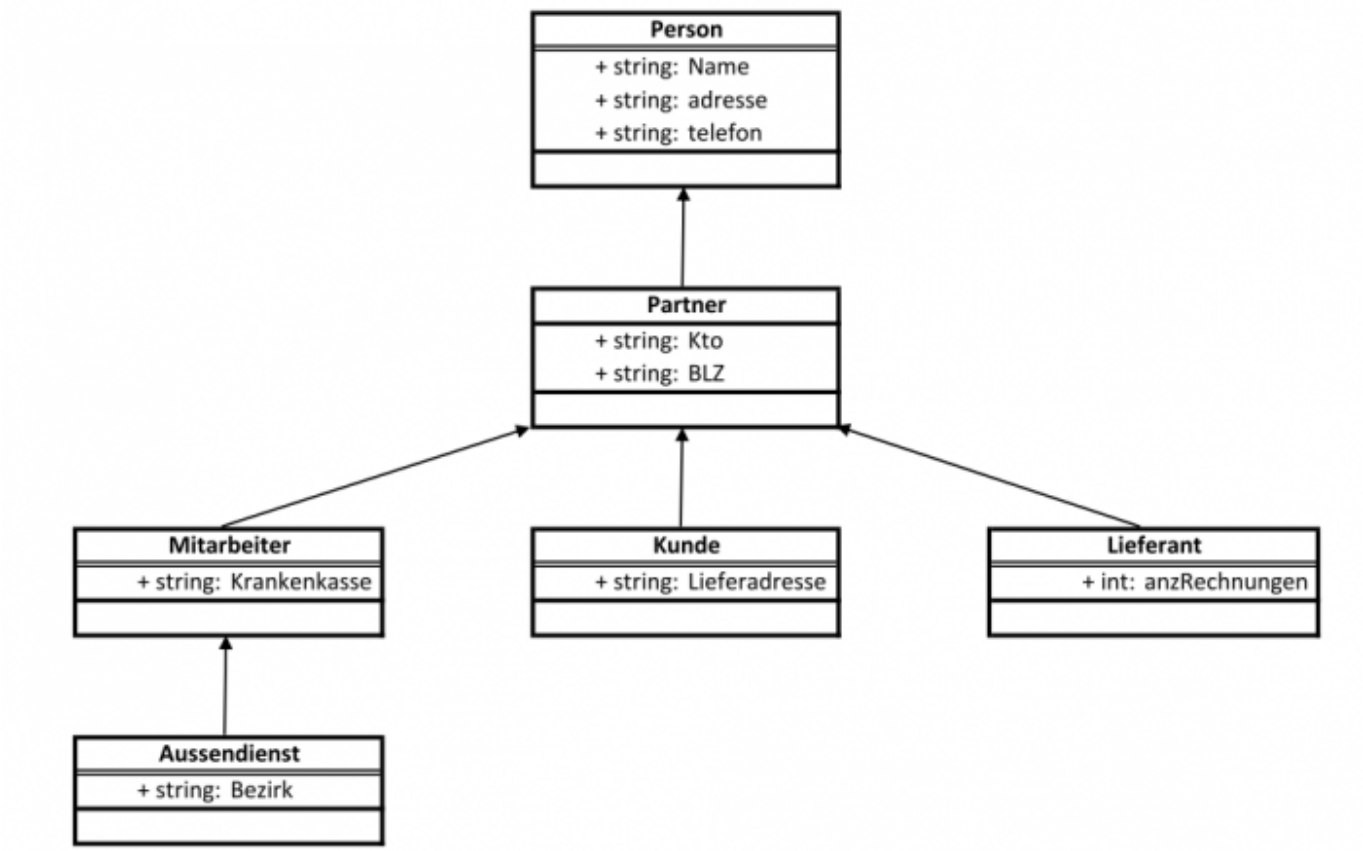
Selbst **Mitarbeiter** sind eigentlich Geschäftspartner, denn sie haben eine Bankverbindung. Darüber hinaus haben sie eine

- Krankenkasse.

Außendienstler haben alle Eigenschaften eines Mitarbeiters und zusätzlich ihren

- Bezirk.

Beispiel als UML-Diagramm:



Beispiel in C + +

```
class Person
{
    public:
       string Name, Adresse, Telefon;
};

class Partner : public Person
{
    public:
       string Kto, BLZ;
};

class Mitarbeiter : public Partner
{
    public:
```

```
       string Krankenkasse;
};

class Kunde : public Partner
{
    public:
       string Lieferadresse;
};

class Lieferant : public Partner
{
    public:
       int anzRechnungen;
};

class Aussendienst : public Mitarbeiter
{
    public:
       string Bezirk;
};

int main (void) {

   Aussendienst a1;

   a1.Bezirk="Amstetten";
   a1.Name="Hans Wurst";
   a1.Krankenkasse="Gebietskrankenkasse";
   a1.BLZ="14200";

   return 0;
}
```

Vorteil von Vererbungen:

Damit stellt die Ausgangsklasse Person die Verallgemeinerung dar und jede abgeleitete Klasse eine Spezialisierung. Der Vorteil dieser Technik ist, dass der bestehende Code der Basisklasse nicht noch einmal für die neu geschaffene Klasse geschrieben werden muss. Beispielsweise würde eine Prüffunktion der Adresse, die der Klasse Person hinzugefügt wird, automatisch auch allen anderen Klassen, die direkt oder indirekt von Person abgeleitet wurden, hinzugefügt, ohne dass eine Zeile Code mehr geschrieben werden müsste. Eine Änderung in der Klasse Mitarbeiter würde immer auch auf die Außendienstler durchschlagen. Es gibt aber keine Rückwirkung auf die Geschäftspartner.

Zusammenfassung

Die objektorientierte Programmierung hat einige neue Begriffe aufgebracht.

Objekt und Klasse

Der zentrale Begriff des Objekts bezeichnet einen Speicherbereich, der durch eine Klasse beschrieben wird. Prinzipiell kann sich der Anfänger ein Objekt als eine besondere Art einer Variablen vorstellen und die Klasse als die Typbeschreibung. Im Buch wird ein Objekt auch hin und wieder als Variable bezeichnet, insbesondere dann, wenn sich ein Objekt an dieser Stelle wie jede andere Variable verhält.

Attribut

Die Daten-Elemente einer Klasse werden in diesem Buch meist Elementvariable genannt. In der objektorientierten Literatur findet sich dafür auch die Bezeichnung Attribut. Damit wird angedeutet, dass die Elementvariablen die Eigenschaften eines Objekts beschreiben.

Methode und Operation

Die Funktionen einer Klasse, die hier als Elementfunktionen bezeichnet werden, finden sich in der objektorientierten Literatur unter dem Namen Methode oder Operation wieder. Diese Bezeichnung bringt zum Ausdruck, dass die Funktion nur über das Objekt erreichbar und damit eine Aktion des Objekts ist.

Konstruktor

Die Elementfunktion, die beim Erzeugen eines Objekts aufgerufen wird, nennt man Konstruktor. In dieser Funktion können Sie dafür sorgen, dass alle Elemente des Objekts korrekt initialisiert sind. Der Konstruktor trägt immer den Namen der Klasse selbst und hat keinen Rückgabetyp, auch nicht void. Der Standardkonstruktor hat keine Parameter.

Dekonstruktor

Im Falle einer Datumsklasse wäre es sinnvoll, dass der Konstruktor alle Elemente auf 0 setzt. Daran kann jede Elementfunktion leicht erkennen, dass das Datum noch nicht festgelegt wurde. Sie könnten alternativ das aktuelle Datum ermitteln und eintragen. Im Beispiel ist auch ein Destruktor definiert worden, obwohl er im Falle eines Datums keine Aufgabe hat

```
/* Beispiel: Klassen (Attribute, Methoden, Konstruktor, Destruktor)
   Filename: main.cpp
   Author: Lahmer
   Title: Klassen
   Description: In diesem Beispiel lernen Sie das Konzept von Klassen
kennen. Es wird eine Klasse Kreis angelegt, welche Attribute (z.B.:
Radius,..), Methoden (z.B.: getFlaeche(),.. ) bzw. Konstruktoren und einen
Destruktor beinhaltet.
   Last Change: 15.02.2018
*/


#include <iostream>
#include <conio.h>
#include <string.h>

using namespace std;

#define PI 3.141


class Kreis {

    private:                                    //Zugriffsrecht private ->
sieht man nur innerhalb der Klasse
        float radius;                           //privates Attribut

    public:                                    //Zugriffsrecht public -> sieht
man auch außerhalb der Klasse
        string beschreibung;                   //öffentliches Attribut
        string einheit;


        /*** Konstruktoren (gleicher Name wie Klasse & kein Rückgabewert)
sind spezielle Methoden ***/
        Kreis()                          // der Default-Konstruktor (wird
standardmäßig ausgeführt)
        {
            radius=1.0;
            einheit="cm";
        }

        Kreis(float r, string e)                    // weiterer Konstruktor
mit Parameter und Defaultwert
        {
            radius=r;
            einheit=e;
        }

        /*** Methoden einer KLasse ***/
        float getRadius()                       //Inline-Methode (Deklaration +
```

```
Definition) getRadius -> gibt den privaten Radius zurück
        {
            return radius;
        }
        float setRadius(float r)            //Inline-Methode (Deklaration +
Definition) setRadius -> setzt den privaten Radius auf r
        {
            radius=r;
        }

        float getUmfang()                    //Inline-Methode (Deklaration +
Definition) getUmfang berechnet den Umfang und gibt diesen zurück
        {
            return 2*radius*PI;
        }

        float getFlaeche();                    //Methodendeklaration
getFlaeche berechnet die Fläche und gibt diese zurück
};

float Kreis::getFlaeche()                  //Methodendefinition der Methode
getFlaeche von der Klasse Kreis
{
    return 2*radius*radius;
}

//Funktionsdefinition
int kreisvergleich(Kreis a, Kreis b)
{
    float kreis1, kreis2;
    kreis1=a.getRadius();
    kreis2=b.getRadius();
    if(a.einheit=="cm")
    {
        kreis1=a.getRadius()/100;
    }
    if(b.einheit=="cm")
    {
        kreis2=b.getRadius()/100;
    }

    if(kreis1>kreis2)
    {
        return 1;
    }
    if(kreis1<kreis2)
    {
        return -1;
    }
    if(kreis1==kreis2)
```

```
    {
        return 0;
    }
}



//Hauptprogramm
int main()
{
    Kreis k1;                    //Objekt
    Kreis k2(2.0, "m");

    //Zugriff auf das öffentliches Attribut beschreibung
    k1.beschreibung="Kreis initialisiert mit Standardkonstruktor!";
    k2.beschreibung="Kreis initialisiert mit Konstruktor mit Radius als
Parameter";

    cout << "KREIS 1:" << endl;
    //Zugriff auf die öffentliche Methode getRadius und das öffentliche
Attribut einheit von k1
    cout << "Radius: \t" << k1.getRadius() << k1.einheit << endl;
    //Zugriff auf die öffentliche Methode getFlaeche und das öffentliche
Attribut einheit von k1
    cout << "Flaeche: \t" << k1.getFlaeche() << k1.einheit << "^2" << endl;
    //Zugriff auf die öffentliche Methode getUmfang und das öffentliche
Attribut einheit von k1
    cout << "Umfang: \t" << k1.getUmfang() << k1.einheit  << endl;

    cout << endl;

    cout << "KREIS 2:" << endl;
    //Zugriff auf die öffentliche Methode getRadius und das öffentliche
Attribut einheit von k2
    cout << "Radius: \t" << k2.getRadius() << k2.einheit  << endl;
    //Zugriff auf die öffentliche Methode getFlaeche und das öffentliche
Attribut einheit von k2
    cout << "Flaeche: \t" << k2.getFlaeche() << k2.einheit  << "^2" << endl;
    //Zugriff auf die öffentliche Methode getUmfang und das öffentliche
Attribut einheit von k2
    cout << "Umfang: \t" << k2.getUmfang() << k2.einheit  << endl;

    float r=0.0;
    cout << endl;
    cout << "Setzen des Radius von Kreis 1:";
    cin >> r;
    k1.setRadius(r);

    r=0.0;
    cout << endl;
    cout << "Setzen des Radius von Kreis 2:";
```

```
    cin >> r;
    k2.setRadius(r);

    cout << endl;

    cout << "KREIS 1:" << endl;
    //Zugriff auf die öffentliche Methode getRadius und das öffentliche
Attribut einheit von k1
    cout << "Radius: \t" << k1.getRadius() << k1.einheit << endl;
    //Zugriff auf die öffentliche Methode getFlaeche und das öffentliche
Attribut einheit von k1
    cout << "Flaeche: \t" << k1.getFlaeche() << k1.einheit << "^2"<< endl;
    //Zugriff auf die öffentliche Methode getUmfang und das öffentliche
Attribut einheit von k1
    cout << "Umfang: \t" << k1.getUmfang() << k1.einheit  << endl;

    cout << endl;

    cout << "KREIS 2:" << endl;
    //Zugriff auf die öffentliche Methode getRadius und das öffentliche
Attribut einheit von k2
    cout << "Radius: \t" << k2.getRadius() << k2.einheit  << endl;
    //Zugriff auf die öffentliche Methode getFlaeche und das öffentliche
Attribut einheit von k2
    cout << "Flaeche: \t" << k2.getFlaeche() << k2.einheit << "^2" << endl;
    //Zugriff auf die öffentliche Methode getUmfang und das öffentliche
Attribut einheit von k2
    cout << "Umfang: \t" << k2.getUmfang() << k2.einheit  << endl;

    cout << "Welcher Kreis ist groesser?" << endl;
    if(kreisvergleich(k1,k2)==1)
    {
        cout << "Der Kreis k1 ist groesser!" << endl;
    }
    if(kreisvergleich(k1, k2)==-1)
    {
        cout << "Der Kreis k2 ist groesser!" << endl;
    }
    if(kreisvergleich(k1, k2)==0)
    {
        cout << "Die beiden Kreise sind gleich gross" << endl;
    }

    return 0;
}
```

DATEIBEHANDLUNG

Programme speichern ihre Informationen in Variablen. Leider bleiben diese immer nur bis zum nächsten Stromausfall oder Betriebssystemabsturz erhalten. Und wenn das Programm verlassen wird, ist ihr Inhalt ebenfalls Geschichte. Damit Sie auf Ihre Daten auch morgen noch kraftvoll zugreifen können, empfiehlt es sich, diese in einer Datei abzulegen. Dazu können Sie die Daten als Ausgabestrom in die Datei schreiben. Das Vorgehen entspricht dem bei der Bildschirmausgabe per cout. Diese Form wird sequenziell genannt, weil die Daten nacheinander in der Reihenfolge, wie sie geschrieben wurden, in der Datei landen. Sie können aber auch einen Datenblock an eine beliebige Stelle der Datei schreiben. Später können Sie diesen Datenblock wieder zurückholen, indem Sie den internen Dateizeiger an diese Stelle positionieren und den Datenblock wieder lesen. Diese Vorgehensweise ist typisch für Klassen, insbesondere, wenn sie in irgendeiner Form sortiert abgelegt werden sollen.

fstream

Über die Headerdatei fstream wird die Funktionalität zum Lesen und Schreiben von Dateien zur Verfügung gestellt. Soll nur geschrieben werden, dann kann auch die Headerdatei ofstream genutzt werden. Ebenso kann, wenn nur gelesen werden soll, als Header ifstream zum Einsatz kommen. Der übliche Ablauf zum Lesen oder Schreiben von Dateien ist folgender:

- Objekt zum Lesen oder Schreiben im Programm anlegen
- Objekt mit dem Dateinamen zum öffnen verknüpfen
- Text über das Objekt schreiben oder lesen
- Datei schliessen

Schreiben einer Datei

In Zeile 5 wird ein Objekt angelegt, dass ähnlich wie cout oder cin die Ausgabe in die Datei übernimmt. Der Name des Objekts kann beliebig vergeben werden, so lange dieser noch nicht verwendet wird. Die Datei beispiel.txt wird in Zeile 6 geöffnet und im aktuellen Verzeichnis angelegt, falls diese noch nicht existiert. Nun da die Datei geöffnet wurde, kann in Zeile 7 etwas in die Datei geschrieben werden. Zum Schluss muss noch die Datei geschlossen werden, damit alle gepufferten Schreibvorgänge abgeschlossen werden (Zeile 8).

```
1  #include <fstream>
2  using namespace std;
3
4  int main () {
5      ofstream fileout;
6      fileout.open("beispiel.txt");
7      fileout << "Das steht jetzt in der Datei.";
8      fileout.close();
9      return 0;
10 }
```

Lesen einer Datei

Nun wollen wir den gerade geschriebenen Inhalt der Datei wieder auslesen und auf dem Bildschirm anzeigen.

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  using namespace std ;
5
6  int main() {
7     string dateizeile;
8     ifstream fin;
9     fin.open("beispiel.txt");
10    getline(fin,dateizeile);
11    fin.close () ;
12    cout << "Zeile in der Datei: " << dateizeile << endl ;
13    return 0;
14 }
```

Wir benötigen einen String dateizeile, in dem wir die gelesene Zeile aus der Datei aufbewahren können. Eine komplette Zeile lässt sich mittels der Funktion getline einlesen. Es wird das mit der Datei verknüpfte Objekt und der String, in den diese Zeile gespeichert werden, übergeben (Zeile 10). Nach dem Schliessen der Datei wird der gelesene String in Zeile 12 ausgegeben.

```
/* Beispiel: Dateibehandlung + Stringfunktionen
   Filename: main.cpp
   Author: Lahmer
   Title: Dateibehandlung
   Description: In diesem Beispiel lernen Sie das Schreiben und Lesen einer
Datei in C++. Zusätzlich lernen Sie verschiedene Stringfunktionen kennen.
   Last Change: 14.02.2018
*/

#include <iostream>
#include <fstream>

using namespace std;

/* run this program using the console pauser or add your own getch,
system("pause") or input loop */


int main(int argc, char** argv) {


    /*** IN DATEI SCHREIBEN ***/
    ofstream write;

    write.open("highscore.csv");
    if(write.is_open())
    {
        write << "********* HIGHSCORE *********\n";
        write << "Andreas;90;m\n";
        write << "Stefan;91;m\n";
        write << "Birgit;85;w\n";
        write << "Lukas;80;m\n";
        write.close();
    }
    else
    {
        cout << "Datei kann nicht geoeffnet werden!\n";
    }


    /*** VON DATEI LESEN ***/
    ifstream read;
    string line="";
    string name="";
    string punkte="";
    string geschlecht="";
    int counter=0;


    read.open("highscore.csv");
```

```
    if(read.is_open())
    {
        //getline liefert solange eine Zeile, bis man am Ende angelangt ist
        while(getline(read, line))
        {
            if(counter>0)
            {
                //Andreas;90;
                name=line.substr(0,line.find(';'));    //Andreas
                punkte=line.substr(line.find(';')+1,line.length());  //90;m;
                punkte=punkte.substr(0, punkte.find(';')); //90
                geschlecht=line.substr(line.length()-1, line.length());
                cout << name << "\t" << punkte << "\t" << geschlecht <<
endl; //Andreas 90 m
            }
            else {
                cout << "######## H I G H S C O R E ###########" << endl;
            }
            counter++;
        }
        read.close();
    }
    else
    {
        cout << "Datei kann nicht geoeffnet werden!\n";
    }
    return 0;
}
```