



Technisches Programmieren in C++


Objekt-Typen

 Karl Riedling
 Institut für Sensor- und Aktuatorssysteme



Objekt-Typen


- Fundamentale Typen
- Direkt abgeleitete Typen
- Zusammengesetzte abgeleitete Typen
- Benutzerdefinierte Typen
- Umwandlungen zwischen Typen
- Der Operator `sizeof`

 Karl Riedling: Technisches Programmieren in C++
 Objekt-Typen

2

Objekt-Typen


- **Fundamentale Typen**
- Direkt abgeleitete Typen
- Zusammengesetzte abgeleitete Typen
- Benutzerdefinierte Typen
- Umwandlungen zwischen Typen
- Der Operator `sizeof`

 Karl Riedling: Technisches Programmieren in C++
 Objekt-Typen

3

Fundamentale Typen


- Fundamentale Datentypen sind mit der Programmiersprache definiert.
- Daten, die diesen Typen entsprechen, werden als "*Variable*" bezeichnet.

 Karl Riedling: Technisches Programmieren in C++
 Objekt-Typen

4

Objekt-Typen

- Fundamentale Typen
 - **Type `void`**
 - Ganzzahlige Typen
 - Gleitkomma-Typen

 Karl Riedling: Technisches Programmieren in C++
 Objekt-Typen


5

Type `void`

- Die Type `void` hat einen leeren Satz von Eigenschaften.
- Sie darf *nicht* für die Deklaration von Variablen verwendet werden.
- Für `void` existieren zwei Anwendungen:
 - Deklaration von Funktionen, die kein Ergebnis zurückgeben:


```
void "C" abort (void);
```
 - "Generische" Zeiger auf Daten mit beliebigen oder nicht festgelegten Typen:


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

 Karl Riedling: Technisches Programmieren in C++
 Objekt-Typen

6

Objekt-Typen

- Fundamentale Typen
 - Type void
 - **Ganzzahlige Typen**
 - Gleitkomma-Typen



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

7

Ganzzahlige Typen

- char
- short (oder short int)
- int
- long (oder long int)



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

8

Ganzzahlige Typen — char

- Type char:
 - Die Type char ist so definiert, dass sie die Elemente des Ausführungs-Zeichensatzes (i.a. ASCII) darzustellen erlaubt.
 - char entspricht daher meistens, aber nicht notwendigerweise, einem Byte zu 8 Bit.
 - Die Type char kann durch Beifügen von signed oder unsigned modifiziert werden.
 - char, signed char und unsigned char werden als *verschiedene* Typen betrachtet.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

9

Ganzzahlige Typen — char

- Type char:
 - In den meisten Implementierungen wird char standardmäßig wie signed char behandelt.
 - Compiler-Optionen erlauben jedoch eine Behandlung von char wie unsigned char.
 - Von der Interpretation als signed oder unsigned hängt das Ergebnis von Vergleichsoperationen und der Umwandlung in andere ganzzahlige Datentypen ab:

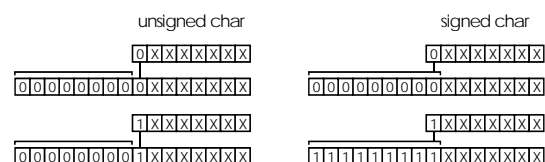


Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

10

Ganzzahlige Typen — char

- Type char:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

11

Ganzzahlige Typen — char

- Beispiel: 8-Bit-char, 16-Bit-short int:

```
signed char ch1 = '\x01', ch2 = '\x81';
unsigned char ch3 = '\x01', ch4 = '\x81';
short int i;
i = ch2;           // i = 0xff81 = -127
i = ch4;           // i = 0x0081 = 129
i = ch1 > ch2;
// i = 1 ("true"), weil 1 > -127
i = ch3 > ch4;
// i = 0 ("false"), weil 1 < 129
```




Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

12

Ganzzahlige Typen — short


- Type short (oder short int):
 - Die Type short ist größer als oder gleich groß wie die Type char und kleiner oder gleich groß wie die Type int.
 - short kann erweitert werden zu signed short und unsigned short.
 - short (ohne Zusatz) oder short int ist immer identisch zu signed short.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 13

Ganzzahlige Typen — int


- Type int:
 - Die Type int ist größer als oder gleich groß wie die Type short int und kleiner oder gleich groß wie die Type long int.
 - int kann erweitert werden zu signed int und unsigned int.
 - int (ohne Zusatz) ist immer identisch zu signed int.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 14

Ganzzahlige Typen — long


- Type long (oder long int):
 - Die Type long ist größer als oder gleich groß wie die Type int.
 - long kann erweitert werden zu signed long und unsigned long.
 - long (ohne Zusatz) oder long int ist immer identisch zu signed long.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 15

Ganzzahlige Typen

- Die tatsächlichen Größen der ganzzahligen Typen und ihrer Wertebereiche hängen von der Implementierung ab:




Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 16

Ganzzahlige Typen

- 16-Bit-Implementierung:

Type	Bytes	Minimum	Maximum
char	1	-127	127
signed char	1	-127	127
unsigned char	1	0	255
short	2	-32767	32767
unsigned short	2	0	65535
int	2	-32767	32767
unsigned int	2	0	65535
long	4	-2147483647	2147483647
unsigned long	4	0	4294967295




Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 17

Ganzzahlige Typen

- 32-Bit-Implementierung:

Type	Bytes	Minimum	Maximum
char	1	-127	127
signed char	1	-127	127
unsigned char	1	0	255
short	2	-32767	32767
unsigned short	2	0	65535
int	4	-2147483647	2147483647
unsigned int	4	0	4294967295
long	4	-2147483647	2147483647
unsigned long	4	0	4294967295



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 18

Objekt-Typen

- Fundamentale Typen
 - Type `void`
 - Ganzzahlige Typen
 - **Gleitkomma-Typen**



Gleitkomma-Typen

- `float`: Die Type `float` ist die kleinste Gleitkomma-Type.
- `double`: Die Type `double` ist größer oder gleich der Type `float`, aber kleiner oder gleich der Type `long double`.
- `long double`: Die Type `long double` ist größer oder gleich der Type `double`.



Gleitkomma-Typen

- Speicherplatzbedarf, Wertebereich und Auflösung der Gleitkommatypen sind wiederum implementierungsspezifisch:

Type	Bytes	Minimum	Maximum	Auflösung
<code>float</code>	4	1.2e-38	3.4e38	1.2e-7
<code>double</code>	8	2.2e-308	1.8e308	2.2e-16
<code>long double</code>	8 10	2.2e-308 3.4e-4932	1.8e308 1.2e4932	2.2e-16 1.1e-19



Objekt-Typen

- Fundamentale Typen
- **Direkt abgeleitete Typen**
- Zusammengesetzte abgeleitete Typen
- Benutzerdefinierte Typen
- Umwandlungen zwischen Typen
- Der Operator `sizeof`



Objekt-Typen

- Direkt abgeleitete Typen
 - **Felder von Variablen oder Objekten (Arrays)**
 - Funktionen
 - Zeiger (*Pointers*)
 - Referenzen (*References*)



Felder (Arrays)

- Felder (*Arrays*) enthalten eine bei ihrer Deklaration festgelegte Anzahl von Elementen *eines bestimmten* Typs, der entweder fundamental, abgeleitet oder benutzerdefiniert sein kann.

- Definition eines Feldes:

```
int IntArray[5];
// Feld aus 5 ints
POINT polygon[6];
// Feld aus 6 Elementen vom
// (Klassen-)Typ POINT
```



Felder (Arrays)

- Feldindex: ganzzahlige Konstante, Variable oder Ausdruck mit ganzzahligem Ergebnis.
- Feldindizes beginnen immer bei 0.
- Das oben deklarierte Feld `IntArray` besteht daher aus den Elementen

```
IntArray [0]
IntArray [1]
IntArray [2]
IntArray [3]
IntArray [4]
```



Felder (Arrays)

- C++ prüft standardmäßig *nicht* die Indizes eines Feldes auf einen gültigen Wertebereich.
- Bei Feldern, deren Größe durch die Initialisierung vorgegeben ist, braucht bei ihrer Definition ihre Größe nicht explizit angegeben zu werden:

```
int primes_to_10[] = {2,3,5,7};
char meldung[] = "Alle Dateien "
"nicht löschen (J/N)?"
```



Felder (Arrays)

- Feld von *Strings* (`char *`):
- ```
char *Bundesland[] = {
 "Wien",
 "Niederösterreich",
 "Burgenland",
 "Oberösterreich",
 "Salzburg",
 "Tirol",
 "Vorarlberg",
 "Steiermark",
 "Kärnten"
};
```



## Felder (Arrays)

- Mehrdimensionale Felder werden als Felder von Feldern interpretiert
- Bei mehrdimensionalen Feldern variiert der letzte Index am schnellsten.



## Felder (Arrays)

- Beispiel für ein zweidimensionales Feld:
- ```
int Feld2d [5][7];
```

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6



Objekt-Typen

- Direkt abgeleitete Typen
 - Felder von Variablen oder Objekten (*Arrays*)
 - **Funktionen**
 - Zeiger (*Pointers*)
 - Referenzen (*References*)



Funktionen

- Funktionen übernehmen eine beliebige Anzahl (einschließlich 0) von *Argumenten* und geben *ein* Resultat zurück (oder keines, wenn sie vom Typ `void` sind).
- Der *Typ* einer *Funktion* ist der Typ des Resultats der Funktion.
- *Funktionen* können daher überall dort verwendet werden, wo auch eine *Konstante* mit dem entsprechenden Typ eingesetzt werden könnte.
- Eine Funktion, die *nicht* vom Typ `void` ist, muss in jedem Fall einen ihrem Typ entsprechenden Wert mit einem `return`-Befehl zurückgeben.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

31

Funktionen

- Beispiel: Funktion, die für ein positives Argument `+1.`, für ein negatives `-1.` und für Argument `==0` `0.` zurückgibt:

```
double sign (double x)
{
    if (x > 0.) // positives Argument
        return 1.;
    else
        if (x < 0.) // negatives Argument
            return -1.;
        else // Argument == 0
            return 0.;
}
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

32

Funktionen

- Sowohl bei der *Deklaration* einer Funktion (*Funktions-Prototyp*) als auch bei ihrer *Definition* müssen Anzahl und Typen der Argumente sowie der Typ des Resultats konsistent angegeben werden.
- Der Compiler prüft bei jedem Aufruf einer Funktion, ob die Typen der Argumente und des Ergebnisses mit der Deklaration übereinstimmen.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

33

Funktionen

- Deklaration; erforderlich, wenn auf die Funktion vor ihrer Definition zugegriffen werden soll:

```
double Power (double Arg, unsigned Exp);
// Funktions-Prototyp; die Namen der
// Argumente können auch weggelassen
// werden:
// double Power (double, unsigned);
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

34

Funktionen

- Definition:

```
double Power (double Arg, unsigned Exp)
{
    double Resultat;
    for (Resultat = 1.; Exp > 0; Exp--)
        Resultat *= Arg;
    return Resultat;
}
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

35

Funktionen

- Beispiel:

```
#include <iostream.h> // für cout und cin
int main ()
{
    double x = 0;
    // muss hier deklariert werden, weil
    // außerhalb des folgenden Blocks
    // (in "while ...") benötigt
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

36

Funktionen

■ Beispiel (Fortsetzung):

```
do
{
    int exp; // kann auch innerhalb der
            // Schleife deklariert werden
    cout << "\nx = "; // Ausgabe
    cin >> x; // Eingabe
    cout << "Exp = ";
    cin >> exp;
    cout << x << "^" << exp << " = " <<
        Power(x, exp) << "\n";
} while (x != 0.);
}
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

37

Objekt-Typen

■ Direkt abgeleitete Typen

- Felder von Variablen oder Objekten (*Arrays*)
- Funktionen
- Zeiger (*Pointers*)**
- Referenzen (*References*)



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

38

Zeiger (*Pointers*)

- Zeiger (*Pointer*) geben die *Adresse* eines Objekts (einer Variablen) vom entsprechenden Typ an.
- Die Adresse einer fundamentalen Variablen oder eines einfachen Objekts kann einem Zeiger unter Verwendung des Operators "&" zugewiesen werden.
- Zeiger auf eine fundamentale Variable:

```
double wert;
double *wp = &wert;
// wp ist ein Zeiger auf eine
// Variable vom Typ double.
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

39

Zeiger (*Pointers*)

- Zeiger auf ein Objekt der Type POINT:

```
POINT Start;
POINT *StartPtr = &Start;
// StartPtr ist ein Zeiger auf ein
// (beliebig komplexes) Objekt vom
// Typ POINT.
```

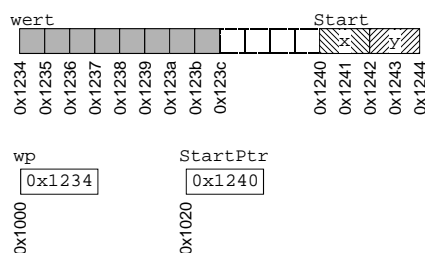


Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

40

Zeiger (*Pointers*)

- Zeiger und Objekte:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

41

Zeiger (*Pointers*)

- Zeiger erlauben die effiziente Übergabe von umfangreichen Objekten als Argumente von Unterprogrammen.
- Übergabe eines Zeigers als Argument erlaubt die Änderung des Wertes des Arguments durch die Funktion.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

42

Zeiger (Pointers)

- Wenn ein Zeiger auf den Beginn eines Feldes (z.B. eines Strings) zeigen soll, erübrigt sich die Angabe des "&"-Operators:

```
char buffer[80];
char *bufptr1 = buffer;
```

- Ein Zeiger auf ein beliebiges Element eines Feldes kann folgendermaßen erhalten werden:

```
char *bufptr2 = &(buffer[4]);
// zeigt auf das 5. Element
char *bufptr3 = buffer + 4;
// äquivalent zu obigem
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

43

Zeiger (Pointers)

- Zeiger auf Feldelemente:

buffer																			
T	h	e	q	u	i	c	k	b	r	o	w	n	f	o	x				
0x1234	0x1235	0x1236	0x1237	0x1238	0x1239	0x123a	0x123b	0x123c	0x123d	0x123e	0x123f	0x1240	0x1241	0x1242	0x1243	0x1244	0x1245	0x1246	0x1247

bufptr1		bufptr2		bufptr3	
0x2000	0x1234	0x2010	0x1238	0x2020	0x1238



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

44

Zeiger (Pointers)

- Der Compiler multipliziert den angegebenen Offset mit der Größe des Objekts (`sizeof (...)`) und addiert ihn zur Basisadresse, um die tatsächliche Adresse im Arbeitsspeicher zu ermitteln.
- Um den Wert der Variablen oder des Objekts zu erhalten, auf die bzw. das der Zeiger weist, muss der Zeiger mit dem Operator "*" dereferenziert werden:

```
char ch1 = *bufptr1; // ch1 = 'T'
char ch2 = *bufptr2; // ch2 = 'q'
```

- Das folgende Programm gibt das char-Feld `buffer` zeichenweise aus:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

45

Zeiger (Pointers)

```
#include <iostream.h> // für cout
char buffer[80] = "The quick brown fox "
                "jumps over the lazy white dog";
int main ()
{
    char *bufptr = buffer;
    while (*bufptr)
        cout << *bufptr++;
    return 0;
}
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

46

Zeiger (Pointers)

- Zeiger gleichen Typs können voneinander subtrahiert werden (z.B. \approx Länge eines Strings).
- Addition/Subtraktion von ganzzahligen Werten zu/von Zeigern \approx Änderung der Zieladresse; Einheit: Elemente der Type des Zeigers.
- Zeiger und Felder werden daher vielfach äquivalent behandelt.
- Der Name eines Feldes *ist de facto* ein Zeiger auf das erste Element des Feldes.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

47

Zeiger (Pointers)

- Zeigerarithmetik:


```
int IntFeld [100];
int *intptr = IntFeld;
                // => IntFeld[0]
intptr += 3; // => IntFeld[3]
```
- Je nach der Größe von `int` (`sizeof (int)`) erhöht sich der Wert von `intptr` im obigen Beispiel um 6 oder 12!



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

48

Zeiger (Pointers)

- Über einen Zeiger kann grundsätzlich das Objekt, auf das er zeigt, geändert werden.
- Um *unerwünschte* Änderungen zu vermeiden, sollte das Schlüsselwort `const` bei der Deklaration des Zeigers verwendet werden:



Zeiger (Pointers)

```
const char *cptr1;
// Zeiger auf eine konstante Variable
char * const cptr2;
// Konstanter Zeiger (zeigt immer auf
// die gleiche char-Variable)
const char * const cptr3;
// Konstanter Zeiger auf konstante
// Variable
```



Zeiger (Pointers)

- Der Compiler lässt nur Zuweisungen zu, die der Deklaration mit `const` nicht widersprechen:

```
const char cch = 'A'; // bleibt immer 'A'
char ch = 'B'; // kann geändert werden
char chl = 'C'; // kann geändert werden
```



Zeiger (Pointers)

```
char *      pch1 = &ch; // erlaubt
const char * pch2 = &ch; // erlaubt
char * const pch3 = &ch; // erlaubt
const char * const pch4 = &ch; // erlaubt
char *      pch5 = &cch; // verboten
const char * pch6 = &cch; // erlaubt
char * const pch7 = &cch; // verboten
const char * const pch8 = &cch; // erlaubt
```



Zeiger (Pointers)

```
*pch1 = 'A'; // Ok; normaler Zeiger
pch1 = &chl; // Ok; normaler Zeiger
*pch2 = 'A'; // Fehler; konstantes Objekt
pch2 = &chl; // Ok; Zeiger nicht konstant
*pch3 = 'A'; // Ok; Objekt nicht konstant
pch3 = &chl; // Fehler; konstanter Zeiger
*pch4 = 'A'; // Fehler; konstantes Objekt
pch4 = &chl; // Fehler; konstanter Zeiger
```



Zeiger (Pointers)

```
pch1 = pch3;
// Ok; nicht konstanter Zeiger darf
// einem konstanten gleichgesetzt
// werden
pch3 = pch1;
// Fehler; ein konstanter Zeiger darf
// nicht einem nicht konstanten gleich
// gesetzt werden.
```



Zeiger (Pointers)

- Bei Funktionsaufrufen mit Zeiger-Argumenten kann das Schlüsselwort `const` unerwünschte Veränderungen von Objekten verhindern.
- `const` als Argument-Attribut wirkt nach innen und außen:



Zeiger (Pointers)

- Funktionsdefinition:

```
void func1 (char *buf1, const char *buf2)
{
    *buf1 = 'A'; // legal; buf1 ist nicht
                // const
    *buf2 = 'B'; // Fehler: Objekt, auf
                // das buf2 weist, darf
                // nicht geändert werden
}
```



Zeiger (Pointers)

- Funktionsaufrufe:

```
char buffer[80];
func1 (buffer, buffer);
// Ok; nicht konstantes Objekt darf in
// konstantes umgewandelt werden
func1 ("Hello, world", buffer);
// Fehler; konstantes Objekt darf
// nicht in nicht konstantes
// umgewandelt werden
func1 (buffer, "Hello, world"); // Ok
```



Zeiger (Pointers)

- In C++ (und auch in C) existieren auch *Zeiger auf Funktionen* (= Einsprungadresse der Funktion).
- Anwendungen von Funktionszeigern:
 - Wenn eine Bibliotheksfunktion eine benutzerdefinierte Funktion aufrufen soll;
 - Wenn über einen ganzzahligen Index-Wert schnell eine bestimmte Funktion ausgewählt und ausgeführt werden soll.



Zeiger (Pointers)

- Beispiele:

```
int func1 (double, int, char *);
int func2 (double, int, char *);
int func3 (double, int, char *);
int (*funcptr) (double, int, char *) = func1;
int (*fpararray[])(double, int, char *) =
{
    func1,
    func2,
    func3
};
```



Zeiger (Pointers)

- Die Klammern bei der Definition von `funcptr` und `fpararray` sind wesentlich:
 - `int (*funcptr) (double, int, char *);`
// ist ein FUNKTIONSZEIGER
 - `int *(funcptr) (double, int, char *);`
// ist eine FUNKTION mit Resultat int*
 - `int *funcptr (double, int, char *);`
// ist eine FUNKTION mit Resultat int*



Zeiger (Pointers)

```
int libfunc (int, int (*)(double, int, char*));
// Prototyp einer Bibliotheksfunktion "libfunc"

int irgendwas (int nPar)
{
    double Wert = 3.14;
    int Schalter = 12;
    char *Meldg = "Hello, world!";
    int i;
    // Funktion funcl über Funktionszeiger ausführen:
    i = funcptr (Wert, Schalter, Meldg);
    // äquivalent zu:
    // i = funcl (Wert, Schalter, Meldg);
}
```



Zeiger (Pointers)

```
// alternative Syntax:
i = (*funcptr)(Wert, Schalter, Meldg);
// Funktions-Zeiger als Argument übergeben:
i = libfunc (4711, funcptr);
// Funktion aus Feld der Funktionszeiger auswählen:
i = fpararray [nPar] (Wert, Schalter, Meldg);
}

int libfunc (int nPar1,
             int (*fpUser) (double, int, char*))
{
    return fpUser (6.28, nPar1, "Error");
}
```



Zeiger (Pointers)

- Auswahl einer Funktion aus einem Array von Funktionszeigern ist oft viel effizienter als Programmverzweigungen.



Objekt-Typen

- Direkt abgeleitete Typen
 - Felder von Variablen oder Objekten (*Arrays*)
 - Funktionen
 - Zeiger (*Pointers*)
 - Referenzen (*References*)



Referenzen (References)

- Referenzen (*References*) sind eigentlich *Zeiger*, die aber syntaktisch wie gewöhnliche *Objekte* behandelt werden:

```
int Wert; // gewöhnliche int-Variablen
int *WertPtr = &Wert;
// Zeiger auf Wert

int& WertRef = Wert;
// Referenz auf Wert
```

In WertRef wird programmintern ebenso wie in WertPtr die Adresse von Wert gespeichert.

- Der Zugriff auf die eigentliche Variable ist über die Referenz jedoch einfacher als über den Zeiger.



Referenzen (References)

- Die folgenden drei Zuweisungen haben identische Wirkung (Wert wird auf 3 gesetzt):

```
Wert = 3;
WertRef = 3;
*WertPtr = 3;
```



Referenzen (*References*)

- Referenzen erleichtern — ebenso wie Zeiger — die Übergabe komplexer Objekte an Funktionen:
- Nur die Adresse des Objekts und nicht das Objekt selbst muss übergeben werden.
- Die Deklaration einer Referenz-Type muss in der Regel eine Initialisierung beinhalten.
- Funktionen können auch Referenzen als Resultat zurückgeben.
- Damit ist es in C++ auch möglich, eine Funktion auf der *linken* Seite einer Zuweisung zu verwenden:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

67

Referenzen (References)

```
int& WertFunc ()
{
    static int Wert; // von außen unzugänglich
    return Wert;
}

// Setzen von Wert:
WertFunc () = 123; // setzt Wert auf 123

// Auslesen von Wert:
int i = WertFunc ();
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

68

Objekt-Typen

- Fundamentale Typen
- Direkt abgeleitete Typen
- **Zusammengesetzte abgeleitete Typen**
- Benutzerdefinierte Typen
- Umwandlungen zwischen Typen
- Der Operator `sizeof`



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

69

Objekt-Typen

- Zusammengesetzte abgeleitete Typen
 - **Strukturen (*Structures*)**
 - *Unions*
 - Klassen (*Classes*)
 - Bitfelder (*Bit Fields*)



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

70

Strukturen (*Structures*)

- Strukturen (*Structures*) vereinigen eine beliebige Anzahl von Objekten mit beliebiger Type.
- Beispiel: Struktur für Datums- und Zeit-Informationen:

```
struct DateTime
{
    short Jahr;
    short Monat;
    short Tag;
    short Stunde;
    short Minute;
    short Sekunde;
};
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

71

Strukturen (*Structures*)

- Deklaration von Struktur-Objekten:


```
struct DateTime Gestern;
DateTime Heute;
```
- Im Gegensatz zu C ist in C++ die Verwendung des Schlüsselwortes `struct` bei der Definition von *Objekten* optional.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

72

Strukturen (*Structures*)

- Gemeinsame Deklaration von Type und Objekt:

```
struct DateTime
{
    short Jahr;
    short Monat;
    short Tag;
    short Stunde;
    short Minute;
    short Sekunde;
}
Morgen;
```



Strukturen (*Structures*)

- Der Zugriff auf die Elemente der Struktur erfolgt mittels des Operators ".".
- Jedes der so referenzierten Elemente kann genau so behandelt werden wie ein Objekt der betreffenden Type (z.B. Heute.Jahr wie eine Variable vom Typ short):

```
Heute.Jahr = 2005;
Heute.Monat = 9;
Heute.Tag = 27;
```



Strukturen (*Structures*)

- Strukturen können als Argumente an eine Funktion übergeben oder von dieser zurückgegeben werden:

```
DateTime Vorjahr (DateTime Jetzt)
{
    Jetzt.Jahr--;
    return Jetzt;
}
DateTime LJahr = Vorjahr (Heute);
```



Strukturen (*Structures*)

- Günstiger ist die Übergabe von Referenzen an eine bzw. von einer Funktion oder die Verwendung von Zeigern:

```
■ Referenz:
DateTime& Vorjahr1 (DateTime& Jetzt)
{
    Jetzt.Jahr--;
    return Jetzt;
}
LJahr = Vorjahr1 (Heute);
```



Strukturen (*Structures*)

- Zeiger:

```
DateTime *Vorjahr2 (DateTime *Jetzt)
{
    (*Jetzt).Jahr--;
    return Jetzt;
}
LJahr = *Vorjahr2 (&Heute);
```



Strukturen (*Structures*)

- Mit Ausnahme der Deklaration der Funktion Vorjahr1
DateTime& Vorjahr1 (DateTime& Jetzt);
resultiert bei Verwendung von Referenzen exakt der gleiche Programm-Quellcode wie bei Verwendung der Objekte (Vorjahr):
DateTime Vorjahr (DateTime Jetzt);



Strukturen (Structures)

- Zeiger auf Strukturen müssen explizit *dereferenziert* werden:

```
DateTime Heute;
DateTime *HeutePtr = &Heute;
    // Zeiger auf die Struktur Heute
(*HeutePtr).Jahr = 2005;
```



Strukturen (Structures)

- Zur Vereinfachung wurde der Operator "`->`" eingeführt:

```
HeutePtr->Jahr = 2005;
HeutePtr->Monat = 9;
HeutePtr->Tag = 27;
    // weist der Struktur Heute die
    // gleichen Werte zu wie im ersten
    // Beispiel
```



Strukturen (Structures)

- Strukturen können ihrerseits auch Strukturen, Zeiger oder Felder enthalten:

```
struct POINT
{
    unsigned x;    // x-Koordinate
    unsigned y;    // y-Koordinate
};
struct Polygon
{
    char *name;    // Bezeichnung
    short ecken;  // Eckenzahl (<= 16)
    POINT poly[16]; // Eckpunkte
};
```



Strukturen (Structures)

```
Polygon Fuenfeck; // Objekt
Polygon *pFuenfeck = &Fuenfeck; // Zeiger
Fuenfeck.name = "Pentagramma";
pFuenfeck->name = "Pentagramma";
cout << Fuenfeck.name;
cout << pFuenfeck->name;
char ch1 = *Fuenfeck.name; // ch1 = 'P'
char ch2 = *pFuenfeck->name; // ch2 = 'P'
Fuenfeck.ecken = 5;
pFuenfeck->ecken = 5;
```



Strukturen (Structures)

```
Fuenfeck.poly[1].x = 2; // x-Wert Ecke 2
pFuenfeck->poly[1].x = 2; // x-Wert Ecke 2

// Die folgenden fünf Zeilen setzen den y-
// Wert von Ecke 3. Achtung auf die Klammern!
((Fuenfeck.poly)+2)->y = 4;
((pFuenfeck->poly)+2)->y = 4;
*((Fuenfeck.poly)+2).y = 4;
*((pFuenfeck->poly)+2).y = 4;
*((*pFuenfeck).poly+2).y = 4;
```



Strukturen (Structures)

- Ablage von Strukturelementen >1 Byte auf ganzzahligen Vielfachen der Prozessor-Wortlänge (2 bzw. 4 Bytes bei 16- bzw. 32-Bit-Prozessoren).
- Kann mit der Präprozessor-Direktive `#pragma pack(n)` bei Bedarf geändert werden (Vorsicht: Portabilitätsprobleme!).



Strukturen (*Structures*)

■ Beispiel:

```
struct test
{
    char ch;
    short ival;
    long lval;
};
```



Strukturen (*Structures*)

```
#pragma pack (1)
ch ival lval
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007

#pragma pack (2)
ch lval lval
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007 0x1008

#pragma pack (4)
ch lval lval
0x1000 0x1001 0x1002 0x1003 0x1004 0x1005 0x1006 0x1007 0x1008 0x1009 0x100a 0x100b 0x100c
```



Objekt-Typen

■ Zusammengesetzte abgeleitete Typen

- Strukturen (*Structures*)
- Unions**
- Klassen (*Classes*)
- Bitfelder (*Bit Fields*)



Unions

- *Unions* verwenden den *gleichen* Speicherbereich für alle ihre Elemente.
- Sie können daher zu einem gegebenen Zeitpunkt nur *ein* Datenelement enthalten.



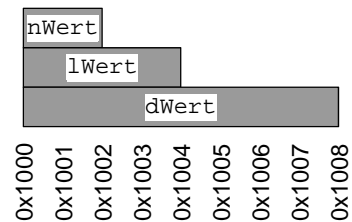
Unions

- Für dieses Datenelement existieren unterschiedliche Interpretationen:

```
union Zahlentype
{
    short nWert;
    long lWert;
    double dWert;
};
```



Unions



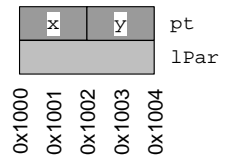
Unions

- Die Verwendung einer Union erlaubt *nicht* die Umwandlung von Werten zwischen verschiedenen Zahlenformaten.
- Der gleiche Speicherplatz wird nur unterschiedlich belegt.
- Beispiel: Zwei Werte vom Typ `short` sollen als ein Wert vom Typ `long` übergeben werden:



Unions

```
struct POINT
{
    short x;
    short y;
}
union LongPar
{
    POINT pt;
    long lPar;
};
```



Unions

```
int WinFunc (int Switch, long Param);
    // Switch bestimmt, wie Param zu
    // interpretieren ist
LongPar Punkt;
Punkt.pt.x = 2;
Punkt.pt.y = 3;
WinFunc (WF_POINT, Punkt.lPar);
```



Objekt-Typen

- Zusammengesetzte abgeleitete Typen
 - Strukturen (*Structures*)
 - Unions
 - Klassen (*Classes*)**
 - Bitfelder (*Bit Fields*)



Klassen (*Classes*)

- Klassen (*Classes*) stellen eine Erweiterung der Funktionalität von Strukturen dar.
- Strukturen *sind* Klassen; ein Objekt, das als `struct` *deklariert* wurde, kann anschließend als `class` *definiert* werden, oder umgekehrt:



Klassen (*Classes*)

```
struct A;    // Voraus-Deklaration
class A     // Definition
{
    public:
        int i;
};
```



Klassen (Classes)

- **Unterschied** zwischen Strukturen und Klassen:
 - Auf die Elemente einer *Struktur* kann standardmäßig von beliebigen Funktionen zugegriffen werden.
 - Auf die Elemente einer *Klasse* können nur Funktionen zugegriffen, die dieser Klasse angehören.



Klassen (Classes)

- Die Schlüsselworte `public` und `private` ändern dieses Verhalten:

```
class X
{
  public:      // alle folgenden Elemente
              // sind frei zugänglich
  int i;
  int j;
  private:   // alle folgenden Elemente
              // sind nur für Funktionen
  int k;
  int l;     // dieser Klasse zugänglich
};
```



Klassen (Classes)

- Der Zugriff auf Datenelemente erfolgt durch *Zugriffsfunktionen*, die Bestandteil der Klasse sind und mit dieser definiert werden.
- Spezielle Klasselement-Funktionen:
 - *Konstruktor*-Funktion;
 - *Destruktor*-Funktion.



Klassen (Classes)

- **Konstruktor**-Funktion:
 - Erstellung und Initialisierung eines neuen Objekts der Klasse.
 - Der Name des Konstruktors ist immer gleich dem Namen der Klasse.
- **Destruktor**-Funktion:
 - "Aufräumen", wenn ein Objekt nicht mehr gültig ist.
 - Der Name des Destruktors ist der Klassenname mit vorangestellter "~".



Klassen (Classes)

```
class Konto // Klassenname "Konto"
{
public:     // frei zugänglich
  Konto () // Konstruktor
  { Kontostand = 0.; }
  // Zugriffsfunktionen:
  void Einlage (double Betrag)
  { Kontostand += Betrag; }
  void Abhebung (double Betrag)
  { Kontostand -= Betrag; }
  double Stand ()
  { return Kontostand; }
private:  // nicht direkt zugänglich
  double Kontostand;
};
```



Klassen (Classes)

```
#include <iostream.h> // für cout

int main ()
{
  Konto Sparbuch; // Bei Erstellung dieses Objekts
                  // wird der Konstruktor ausgeführt
  cout << "Saldo: " << Sparbuch.Stand() << "\n";
                  // gibt "0" aus
  Sparbuch.Einlage (500.);
  Sparbuch.Abhebung (200.);
  cout << "Saldo: " << Sparbuch.Stand() << "\n";
                  // gibt "300" aus
}
```



Klassen (Classes)

- Zugriffsfunktionen sind logischer, aber nicht physikalischer Teil der Klasse.
- Sie müssen (wie ein Struktur-Element) in der Form `Objekt.Funktionsname` aufgerufen werden.
- Zugriffsfunktionen können entweder bei der Deklaration der Klasse oder separat definiert werden.
- Zugriffsfunktionen, die *mit* der Klasse definiert wurden, werden in manchen Implementierungen *immer* als inline-Funktionen angelegt.
- Bei der separaten Definition einer Zugriffsfunktion muss ihrem Namen der Name der Klasse, gefolgt vom Operator `::` (z.B. `Konto::`), vorangestellt werden:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

103

Klassen (Classes)

```
class Konto
{
public:
    Konto (); // Konstruktor, hier nur deklariert
            // Zugriffsfunktionen; nur deklariert:
    void Einlage (double Betrag);
    void Abhebung (double);
    double Stand ();

private:
    double Kontostand;
};
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

104

Klassen (Classes)

```
// Definition der Funktionen:
Konto::Konto ()
{ Kontostand = 0.; }
void Konto::Einlage (double Betrag)
{ Kontostand += Betrag; }
void Konto::Abhebung (double Betrag)
{ Kontostand -= Betrag; }
double Konto::Stand ()
{ return Kontostand; }
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

105

Klassen (Classes)

- Die Datenelemente einer Klasse sind *lokal für jedes Objekt* dieser Klasse, d.h., für jedes Objekt der Klasse `Konto` existiert *genau ein* Datenelement `Kontostand`.
- Wird ein Element in einer Klasse als `static` deklariert, wird *genau ein* derartiges Element (mit externer Gültigkeit) angelegt.
- *Statische Elemente* werden mit der Deklaration einer Klasse *zwar deklariert*, sie müssen aber *explizit definiert* und *initialisiert* werden:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

106

Klassen (Classes)

```
class Punkt
{
public:
    Punkt () { PunkteZahl++; } // Konstruktor
    ~Punkt () { PunkteZahl--; } // Destruktor
            // Zugriffsfunktionen:
    unsigned x() { return xKoord; }
    unsigned y() { return yKoord; }
            // statische Zugriffsfunktion:
    static int Anzahl() { return PunkteZahl; }
    static int PunkteZahl; // statisches Datenelement
private:
    unsigned xKoord; // "gewöhnliche" Datenelemente
    unsigned yKoord;
};
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

107

Klassen (Classes)

```
int Punkt::PunkteZahl = 0; // Definition von PunkteZahl

int main ()
{
    cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
    Punkt p1;
    cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
    {
        Punkt p2;
        cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
    }
    Punkt p3;
    cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
}
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

108

Klassen (Classes)

```
// Zugriffsfunktionen mit Referenz-Ergebnis!
p1.x() = 20;
p1.y() = 30;
}
// Punkt p3 existiert nicht mehr
cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
}
// Punkt p2 existiert nicht mehr
cout << "Anzahl der Punkte: " << Punkt::Anzahl() << "\n";
// Koordinaten von Punkt p1 ausgeben
cout << p1.x() << ", " << p1.y() << "\n";
}
```



Klassen (Classes)

- Statische Klassenelement-Funktionen können auch aufgerufen werden, ohne dass ein Objekt dieser Klasse existiert.
- Sie müssen entweder mit dem Namen der Klasse (z.B. "Punkt::Anzahl()") oder über ein *Objekt* der Klasse (z.B. "p1.Anzahl()") identifiziert werden.
- Gleiches gilt für Zugriffe auf statische *Datenelemente* einer Klasse.



Klassen (Classes)

- Ebenso wie bei Strukturen können *Zeiger auf Objekte* einer Klasse definiert und verwendet werden.
- In C++ existieren *Zeiger auf ein Element der Klasse*. Ein Zeiger auf ein Element einer Klasse ist eine *Type*, kein *Objekt*!



Objekt-Typen

- Zusammengesetzte abgeleitete Typen
 - Strukturen (*Structures*)
 - Unions
 - Klassen (*Classes*)
 - Bitfelder (*Bit Fields*)



Bitfelder (*Bit Fields*)

- Klassen (*struct* oder *class*) können Elemente (Bitfelder, *Bit Fields*) enthalten, die *kleiner* als eine ganzzahlige Datentype sind.
- Deklaration von Bitfeldern:
 - Deklarator* (ganzzahlige Type, optional Name),
 - Doppelpunkt (":"),
 - Ganzzahliger konstanter Ausdruck = Anzahl der Bits in dem Bitfeld.
- *Anonyme* (also namenlose) Bitfelder können als Platzhalter verwendet werden.




Bitfelder (*Bit Fields*)

- Ein Element der Größe 0 erzwingt den Beginn des nächsten Elements in einem neuen Element der ganzzahligen Type.
- Üblicherweise *unsigned*-Typen für Bitfelder (nicht von der Syntax verlangt).
- Typische Anwendung:
 - Hardwarenahe Problemstellungen;
 - Die einzelnen Bits eines Bytes, Wortes oder Doppelwortes müssen unabhängig voneinander behandelt werden.



Bitfelder (Bit Fields)

- Größe des Basistype des Bitfelds (unsigned char, unsigned short oder unsigned long) & Anordnung der Bitfelder.
- Basistype entsprechend der Anwendung wählen (8, 16 oder 32 Bit).

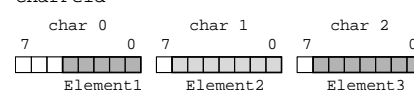



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 115

Bitfelder (Bit Fields)

```
struct CharFeld
{
    unsigned char Element1 : 5;
    unsigned char Element2 : 7;
    unsigned char Element3 : 7;
};
```

CharFeld:






Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 116

Bitfelder (Bit Fields)

```
struct ShortFeld
{
    unsigned short Element1 : 5;
    unsigned short Element2 : 7;
    unsigned short Element3 : 7;
};
```

ShortFeld:

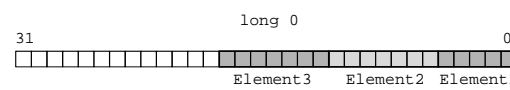




Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 117

Bitfelder (Bit Fields)

```
struct LongFeld
{
    unsigned long Element1 : 5;
    unsigned long Element2 : 7;
    unsigned long Element3 : 7;
};
```

LongFeld:

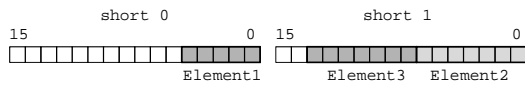




Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 118

Bitfelder (Bit Fields)

```
struct AlignedShortFeld
{
    unsigned short Element1 : 5;
    unsigned short      : 0;
    unsigned short Element2 : 7;
    unsigned short Element3 : 7;
};
```


AlignedShortFeld:

Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 119

Objekt-Typen

- Fundamentale Typen
- Direkt abgeleitete Typen
- Zusammengesetzte abgeleitete Typen
- **Benutzerdefinierte Typen**
- Umwandlungen zwischen Typen
- Der Operator sizeof



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen 120

Objekt-Typen

- Benutzerdefinierte Typen
 - **Aufzählungen (Enumerations)**
 - Synonyme (typedef)



Aufzählungen (Enumerations)

- *Enumerations* sind ganzzahlige Typen, die eine Liste *benannter Konstanten* definieren.
- Aufzählungstypen werden intern als ganze Zahlen dargestellt.
- Standardmäßig wird der erste Name (*Enumerator*) in der Liste als "0" dargestellt, der zweite als "1", usw.



Aufzählungen (Enumerations)

```
enum Tag
{
    Sonntag,           // => 0
    Montag,            // => 1
    Dienstag,         // => 2
    Mittwoch,          // => 3
    Donnerstag,        // => 4
    Freitag,           // => 5
    Samstag            // => 6
};
```



Aufzählungen (Enumerations)

```
int main ()
{
    Tag Heute; // Deklaration eines Objekts
               // "Heute" vom Typ "Tag"

    Heute = Dienstag;
               // Zuweisung eines Wertes aus
               // dem Wertevorrat von "Tag"
}
```



Aufzählungen (Enumerations)

- *Enumerations* können überall dort verwendet werden, wo Konstanten verwendet werden können (also *nicht* als Ergebnis eines Ausdrucks!):

```
Heute = Dienstag;           // Ok
if (Heute == Donnerstag)... // Ok

Tag Morgen;
Morgen = Heute++;           // unzulässig!
```

- Auswege:
 - Bedeutung eines Operators für die Aufzählungstypen neu definieren (*Operator Overloading*);
 - Typumwandlung.



Aufzählungen (Enumerations)

- Die numerischen Werte einer Aufzählung können auch explizit zugewiesen werden:

```
enum Tag1
{
    Sonntag,           // => 0
    Montag = 5,        // => 5
    Dienstag,         // => 6
    Mittwoch,          // => 7
    Donnerstag = 3,    // => 3
    Freitag,           // => 4
    Samstag            // => 5
};
```



Aufzählungen (*Enumerations*)

- Der Name eines Enumerators muss eindeutig sein.
- Enumeratoren haben keine dateiübergreifende Gültigkeit (*Linkage*).
- Es ist also nicht möglich, sich in einem Modul auf einen in einem anderen Modul definierten Enumerator zu beziehen.
- Aufzählungen ersetzen einen *Zahlenwert* durch einen *logischen Namen*.
- Die Verwendung eines Enumerators ist aber auf Objekte beschränkt, die als Aufzählung (unter Verwendung eben dieses Enumerators) definiert wurden.

```
int i = Montag;           // unzulässig
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

127

Aufzählungen (*Enumerations*)

- Alternative Möglichkeiten, Konstante durch Namen zu ersetzen, sind:
 - Präprozessor-Makros:


```
#define MONTAG 1
int i = MONTAG; // ist "int i = 1;"
```
 - Definition mit dem Schlüsselwort `const`:


```
const int montag = 1;
int i = montag; // ist "int i = 1;"
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

128

Objekt-Typen

- Benutzerdefinierte Typen
 - Aufzählungen (*Enumerations*)
 - **Synonyme (*typedef*)**



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

129

Synonyme (*typedef*)

- Das Schlüsselwort `typedef` erlaubt die Zuweisung eines Namens an beliebige (und beliebig komplexe) Datentypen.
- Der mit `typedef` angegebene Name wird zum Namen einer Datentype.
- Er kann überall dort verwendet werden, wo fundamentale oder abgeleitete Typen verwendet werden können:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

130

Synonyme (*typedef*)

```
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef unsigned long DWORD;

typedef BYTE * PBYTE;
// PBYTE entspricht "unsigned char *"

typedef int (*)(double, int, char*) MyType;
// Funktions-Zeiger aus früherem Abschnitt

int libfunc (int, MyType); // steht für:
int libfunc (int, int (*)(double, int, char*));
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

131

Synonyme (*typedef*)

- `typedef`-Namen können in (nachfolgenden) `typedefs` vorkommen.
- Mit `typedef` definierte Synonyme sind nur innerhalb der Datei gültig, in der sie definiert wurden.
- `typedef`-Namen haben keine dateiübergreifende Gültigkeit (*Linkage*).



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

132

Synonyme (typedef)

- In C++ (nicht in C) sind mehrere *in ihrem Wesen* identische typedef-Definitionen in einer Datei zulässig:

```
typedef signed char CHAR;
typedef signed char CHAR;
    // Ok; identische Definition
typedef CHAR CHAR;
    // Ok; kein Widerspruch
typedef char CHAR;
    // Fehler! char != signed char !
```



Synonyme (typedef)

- Die Namen von typedefs müssen sich von denen von Variablen, Funktionen und anderen Objekten unterscheiden.
- Sie können aber durch Deklaration eines Objekts mit gleichem Namen in einem eingeschlossenen Block verborgen werden.



Synonyme (typedef)

- Der folgende (C-spezifische) Code wird aus Kompatibilitätsgründen auch in C++ akzeptiert:

```
typedef struct // namenlose Struktur
{
    short x;
    short y;
}
PUNKT; // benannt als "PUNKT"
PUNKT p1, p2; // Definition von Objekten
```



Objekt-Typen

- Fundamentale Typen
- Direkt abgeleitete Typen
- Zusammengesetzte abgeleitete Typen
- Benutzerdefinierte Typen
- Umwandlungen zwischen Typen**
- Der Operator `sizeof`



Objekt-Typen

- Umwandlungen zwischen Typen
 - Implizite Umwandlungen**
 - Explizite Umwandlungen



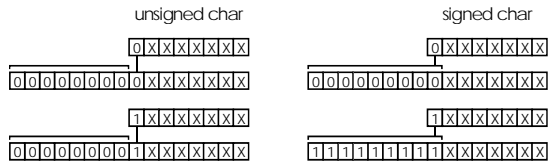
Implizite Umwandlungen

- Automatische Umwandlung in andere Datentypen bei:
 - Arithmetischen Ausdrücken;
 - Übergabe von Argumenten an Funktionen;
 - Zuweisung eines Ergebnisses eines Ausdrucks.
- Integral Promotion*: Umwandlung der folgenden Datentypen in ganzzahlige Datentypen:
 - Objekte vom Typ `char` und `short int`;
 - Aufzählungs-Typen;
 - Bitfelder.



Implizite Umwandlungen

- *Integral Promotion* erfolgt durch "Auffüllen" der hinzugekommenen Bits:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

139

Implizite Umwandlungen

- Die Konversion erfolgt in die Type `int`, sofern `int` geeignet ist, den gesamten Wertevorrat darzustellen; ansonsten erfolgt sie in `unsigned int`.
- Dabei wird der *Wert* (= das Bitmuster), nicht aber unbedingt das *Vorzeichen* erhalten.
- Variable vom Typ `unsigned int` werden in 16-Bit-Systemen in `long` umgewandelt; in 32-Bit-Systemen erfolgt ihre Umwandlung in `unsigned long`.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

140

Implizite Umwandlungen

- Dieses Verhalten kann zu unterschiedlichen Resultaten bei Vergleichsoperationen auf 16- und 32-Bit-Systemen führen:
 - 16-Bit-System: `(-1L < 1U) == 1 (true)`, weil beide Seiten in den Typ (`signed`) `long` umgewandelt werden und `(-1L < +1L)` ist.
 - 32-Bit-System: `(-1L < 1U) == 0 (false)`, weil beide Seiten in `unsigned long` konvertiert werden müssen und `-1UL (= 232 - 1) > +1UL (= 1)` ist.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

141

Implizite Umwandlungen

- Bei der Konversion ganzzahliger Datentypen zwischen `signed` und `unsigned` bleibt grundsätzlich das Bitmuster der Variablen erhalten, nur die Interpretation ändert sich:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

142

Implizite Umwandlungen

```
#include <iostream.h>
int main ()
{
    short i = -3;
    unsigned short u;
    cout << (u = i) << "\n";
    // Wichtig: Klammer um "(u = i)"
    return 0;
}
// Das Programm gibt 65533 (216 - 3) aus.
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

143

Implizite Umwandlungen

```
#include <iostream.h>
int main ()
{
    short i;
    unsigned short u = 65533;
    cout << (i = u) << "\n";
    return 0;
}
// Das Programm gibt die Zahl -3 aus.
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

144

Implizite Umwandlungen

- Konversionen zwischen *Gleitkomma-Typen* sind ohne Verlust von Genauigkeit und Wert möglich, solange die Konversion in eine "höhere" Type erfolgt.
- In umgekehrter Richtung wird die Genauigkeit auf die der Zieltype reduziert.
- Das Ergebnis der Konversion kann für Werte außerhalb des darstellbaren Wertebereichs der Zieltype Null oder Unendlich (`1.#INF`) sein.
- Bei der Konversion von Gleitkommawerten in ganzzahlige Werte wird stets der nichtganzzahlige Teil abgeschnitten (*nicht gerundet!*). Aus `1.9` wird `1`, und aus `-1.9` wird `-1`.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

145

Implizite Umwandlungen

- Bei arithmetischen Operationen mit zwei Operanden erfolgt stets eine Konversion in die "höhere" der beiden beteiligten Typen:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

146

Implizite Umwandlungen

Typ eines Operanden:	Typ des 2. Operanden:	Konversion in:
long double	gleichgültig	long double
double	gleichgültig	double
float	gleichgültig	float
unsigned long	gleichgültig	unsigned long
long	unsigned int	long (16 Bit) unsigned long (32 Bit)
long	nicht unsigned int	long
unsigned int	gleichgültig	unsigned int
alle anderen	gleichgültig	int



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

147

Implizite Umwandlungen

- *Zeiger* beliebiger Typen können implizit in einen Zeiger der Type `void *` konvertiert werden.
- Die umgekehrte Konversion ist nur explizit möglich.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

148

Objekt-Typen

- Umwandlungen zwischen Typen
 - Implizite Umwandlungen
 - **Explizite Umwandlungen**
 - **Type Casts**
 - Typkonversions-Operator



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

149

Type Casts

- *Type Casts* wandeln ein Objekt einer Type in das einer anderen um.
- Sie erhalten den Wert der Objekte, auf die sie angewendet werden (sofern möglich).
- *Type Casts* bestehen aus dem Namen der Type in runden Klammern, der dem zu konvertierenden Objekt vorangestellt wird:

```
int i = 7;
double d;
d = (double) i;
// In C++ wäre hier der Type Cast
// gar nicht nötig gewesen.
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

150

Type Casts

- Explizite Typenumwandlungen sind unbedingt erforderlich für die Konversion von *Zeigern* unterschiedlicher Typen.
- Beispiel: Für einen Speicherbereich `buffer`, der irgendwelche Daten vom Typ `char` enthält, soll eine 16-Bit-Prüfsumme berechnet werden:



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

151

Type Casts

```
unsigned short checksum (char *buffer, unsigned int
    size)
{
    unsigned short sum = 0;           // Prüfsumme
    unsigned int i = size / sizeof(short); // Anzahl der shorts in buffer
    while (i)
    {
        sum += *(unsigned short *) buffer;
        ((unsigned short *) buffer)++;
        i--;
    }
    return sum;
}
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

152

Type Casts

- Damit wird das gleiche Ergebnis erhalten, wie wenn `buffer` als `unsigned short *buffer` definiert worden wäre.
- Die Schleife hätte noch kürzer geschrieben werden können:

```
while (i--)
    sum += *((unsigned short *) buffer)++;
```



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

153

Type Casts

- Größte Vorsicht bei der Verwendung von Typkonversionen, weil sie die Typenprüfung des Compilers unterlaufen!
- Insbesondere die unüberlegte Konversion von Zeigern kann zu unerwarteten Ergebnissen führen!



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

154

Objekt-Typen

- Umwandlungen zwischen Typen
 - Implizite Umwandlungen
 - **Explizite Umwandlungen**
 - *Type Casts*
 - **Typkonversions-Operator**



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

155

Typkonversions-Operator

- Bevorzugte Form der expliziten Typenkonversion in C++:


```
int i = 7;
double d;
d = double (i);
```
- Vorteil gegenüber *Type Casts*: Mehr als ein Argument für die Konversion möglich.



Karl Riedling: Technisches Programmieren in C++
Objekt-Typen

156

Typkonversions-Operator

- Anwendung auf komplexe Typen (z.B. Klassen) mit der gleichen Syntax möglich:

```
struct Point
{
    Point (short x, short y)
    {_x = x; _y = y } // Konstruktor
    short _x, _y;
}
short i, j;
...
Point pt = Point (i, j);
```



Objekt-Typen

- Fundamentale Typen
- Direkt abgeleitete Typen
- Zusammengesetzte abgeleitete Typen
- Benutzerdefinierte Typen
- Umwandlungen zwischen Typen
- Der Operator `sizeof`



Der Operator `sizeof`

- Ergebnis von `sizeof`: Größe des Objekts oder der Type in chars, auf das oder die `sizeof` angewendet wird.
- Aus Portabilitätsgründen Größen von Objekten oder Typen *grundsätzlich* nicht als numerische Konstante, sondern mit `sizeof` angeben.
- `sizeof` wird *immer* bei der Übersetzung des Programms in eine ganzzahlige Konstante mit dem korrekten Wert umgewandelt.



Der Operator `sizeof`

- Größe einer *Type* in Einheiten der Type char: Name der Type *in Klammern*:

```
int i = sizeof (int);
// in 16-Bit-Systemen: "int i = 2;"
// in 32-Bit-Systemen: "int i = 4;"
```

- Größe eines *Objekts* in Einheiten der Type char: Name des Objekts *ohne Klammern*:

```
char szHello[] = "Hello, world!";
int i = sizeof szHello;
// immer: "int i = 14;"
```



Der Operator `sizeof`

- Die Verwendung von `sizeof` erlaubt die Bestimmung der Anzahl der Elemente eines Feldes:


```
int Groesse = sizeof Feld/sizeof Feld[0];
```
- Wenn `sizeof` auf eine Referenz angewendet wird, ist das Resultat dasselbe, wie wenn `sizeof` auf das Objekt selbst angewendet worden wäre.
- `sizeof` darf *nicht* angewendet werden auf:
 - Funktionen (wohl aber auf Zeiger auf Funktionen),
 - Bitfelder,
 - undefinierte Klassen,
 - die Type `void`.

