Datenstrukturen und Datentypen

\\//_

17. Dezember 2020

Gliederung

1. Datentypen und Datenstrukturen
2. einfache Datentypen
   - ordinale Datentypen
   - nicht ordinale Datentypen
3. zusammengesetzte Datentypen
   - Zeichenkette
   - Feld
   - Verbund
   - Datei
   - Baum

Datentypen – Sinn und Zweck

Für das Programmieren und auch das Benutzen von Anwendungsprogrammen ist es von Bedeutung, welcher Art die verwendeten Daten sind.
An folgenden Überlegungen wird das z. B. deutlich:

Datentypen – Sinn und Zweck

Für das Programmieren und auch das Benutzen von Anwendungsprogrammen ist es von Bedeutung, welcher Art die verwendeten Daten sind.
An folgenden Überlegungen wird das z. B. deutlich:

- nur mit Zahlen kann man rechnen (nicht mit Text)
  ➠ Tabellenkalkulation, Datenbanken, . . .

Datentypen – Sinn und Zweck

Für das Programmieren und auch das Benutzen von Anwendungsprogrammen ist es von Bedeutung, welcher Art die verwendeten Daten sind.
An folgenden Überlegungen wird das z. B. deutlich:

- nur mit Zahlen kann man rechnen (nicht mit Text)
  ➠ Tabellenkalkulation, Datenbanken, . . .
- Division mit ganzen oder reellen Zahlen . . .

Datentypen – Sinn und Zweck

Für das Programmieren und auch das Benutzen von Anwendungsprogrammen ist es von Bedeutung, welcher Art die verwendeten Daten sind.
An folgenden Überlegungen wird das z. B. deutlich:

- nur mit Zahlen kann man rechnen (nicht mit Text)
  ➠ Tabellenkalkulation, Datenbanken, . . .
- Division mit ganzen oder reellen Zahlen . . .
- unterschiedlicher Speicherbedarf (Zahlenbereiche)

Datentypen – Sinn und Zweck

Für das Programmieren und auch das Benutzen von Anwendungsprogrammen ist es von Bedeutung, welcher Art die verwendeten Daten sind.
An folgenden Überlegungen wird das z. B. deutlich:

- nur mit Zahlen kann man rechnen (nicht mit Text)
  ➠ Tabellenkalkulation, Datenbanken, . . .
- Division mit ganzen oder reellen Zahlen . . .
- unterschiedlicher Speicherbedarf (Zahlenbereiche)
- Nutzung der Festkomma- oder Gleitkommaeinheit des Prozessors?

Datentypen – Sinn und Zweck

Für das Programmieren und auch das Benutzen von Anwendungsprogrammen ist es von Bedeutung, welcher Art die verwendeten Daten sind.
An folgenden Überlegungen wird das z. B. deutlich:

- nur mit Zahlen kann man rechnen (nicht mit Text)
  ➠ Tabellenkalkulation, Datenbanken, ...
- Division mit ganzen oder reellen Zahlen ...
- unterschiedlicher Speicherbedarf (Zahlenbereiche)
- Nutzung der Festkomma- oder Gleitkommaeinheit des Prozessors?
- logische Operationen, Zählbarkeit, ...

Datentypen und Datenstrukturen

Der Begriff **Datentyp**

. . . beschreibt den Wertebereich von Daten, in dem ganz bestimmte Operationen möglich sind, die auf alle Daten dieses Typs anwendbar sind.

Datentypen und Datenstrukturen

Der Begriff **Datentyp**

. . . beschreibt den Wertebereich von Daten, in dem ganz bestimmte Operationen möglich sind, die auf alle Daten dieses Typs anwendbar sind.

Beispiel – Operationen bei Zahlen-Datentypen:

Datentypen und Datenstrukturen

Der Begriff **Datentyp**

. . . beschreibt den Wertebereich von Daten, in dem ganz bestimmte Operationen möglich sind, die auf alle Daten dieses Typs anwendbar sind.

Beispiel – Operationen bei Zahlen-Datentypen:

- **ganze Zahlen:** $+$ (Addition), $-$ (Subtraktion), $*$ (Multiplikation), *div* (ganzzahlige Division), *mod* (Rest bei ganzzahl. Division), *abs* (Absolutbetrag)

Datentypen und Datenstrukturen

Der Begriff **Datentyp**

. . . beschreibt den Wertebereich von Daten, in dem ganz bestimmte Operationen möglich sind, die auf alle Daten dieses Typs anwendbar sind.

Beispiel – Operationen bei Zahlen-Datentypen:

- **ganze Zahlen:** $+$ (Addition), $-$ (Subtraktion), $*$ (Multiplikation), *div* (ganzzahlige Division), *mod* (Rest bei ganzzahl. Division), *abs* (Absolutbetrag)
- **reelle Zahlen:** $+$ (Addition), $-$ (Subtraktion), $*$ (Multiplikation), / (Division), verschiedene mathematische Funktionen (*log*, *sin*, . . .)

Datentypen und Datenstrukturen

Der Begriff **Datenstruktur**

. . . beschreibt die Zusammenfassung gleicher oder unterschiedlicher Datentypen nach einem bestimmten Konstruktionsprinzip.

Datentypen und Datenstrukturen

Der Begriff **Datenstruktur**

. . . beschreibt die Zusammenfassung gleicher oder unterschiedlicher Datentypen nach einem bestimmten Konstruktionsprinzip.
In Programmiersprachen benutzt man dafür auch den Begriff **„zusammengesetzter“** oder **„strukturierter Datentyp“**.

Datentypen und Datenstrukturen

Der Begriff **Datenstruktur**

. . . beschreibt die Zusammenfassung gleicher oder unterschiedlicher Datentypen nach einem bestimmten Konstruktionsprinzip.
In Programmiersprachen benutzt man dafür auch den Begriff **„zusammengesetzter“** oder **„strukturierter Datentyp“**.

Zu den Datenstrukturen gehören die **Zeichenkette** *(string)*, das **Feld** *(array)*, der **Verbund** *(record)*, die **Datei** *(file)* und der **Baum** *(tree)*.

Datentypen und Datenstrukturen

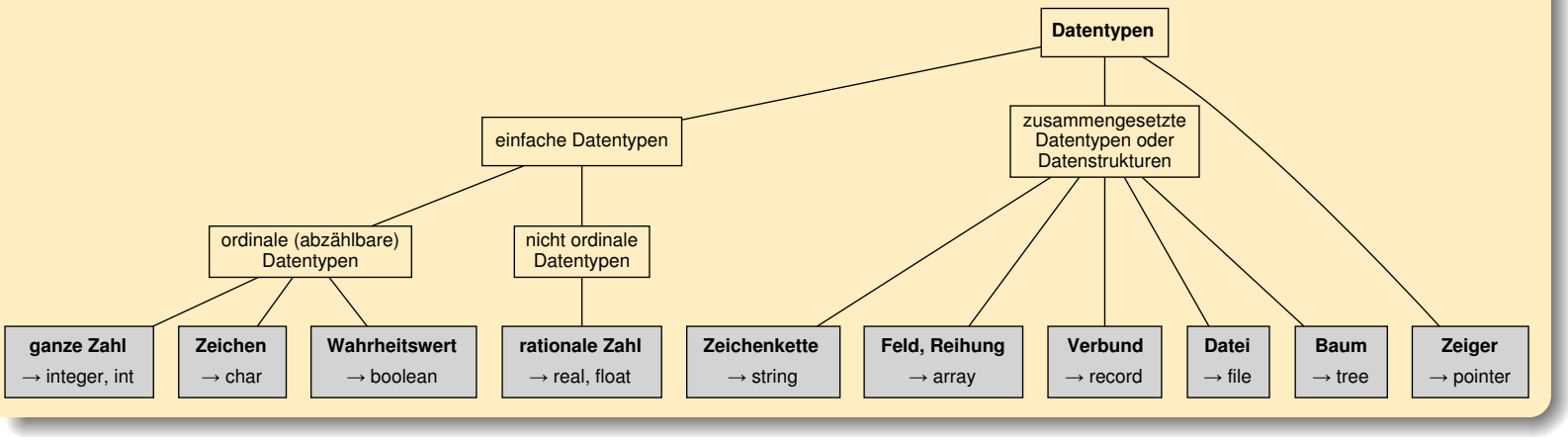
Der Begriff **Datenstruktur**

. . . beschreibt die Zusammenfassung gleicher oder unterschiedlicher Datentypen nach einem bestimmten Konstruktionsprinzip.
In Programmiersprachen benutzt man dafür auch den Begriff **„zusammengesetzter“** oder **„strukturierter Datentyp“**.

Zu den Datenstrukturen gehören die **Zeichenkette** *(string)*, das **Feld** *(array)*, der **Verbund** *(record)*, die **Datei** *(file)* und der **Baum** *(tree)*.
Eine Sonderstellung haben die **Zeiger** *(pointer)*. Sie verweisen auf beliebige andere Datentypen. Daher bleibt ihr wirklicher Wertebereich meist anonym.

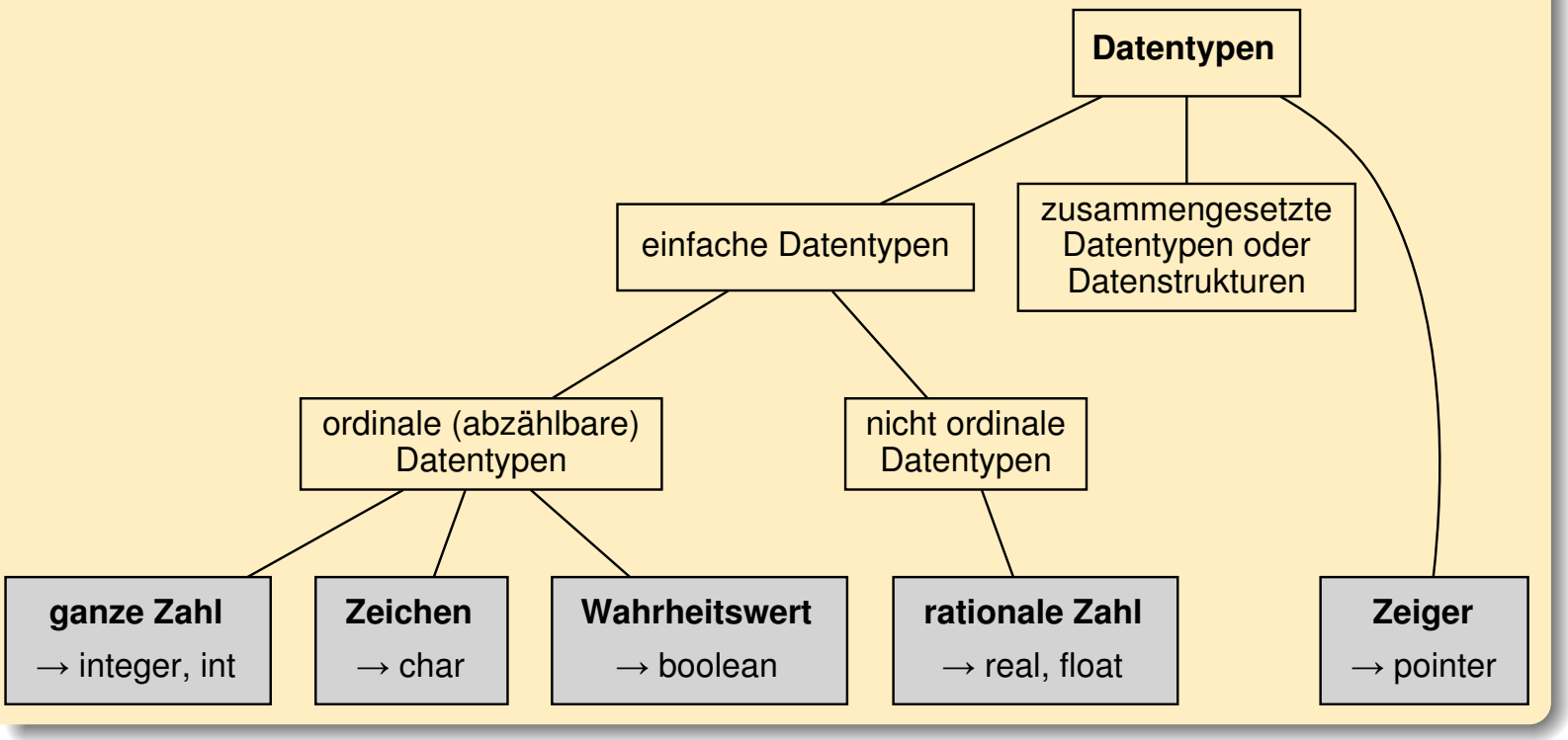
Datentypen und Datenstrukturen

Einteilung (komplett)



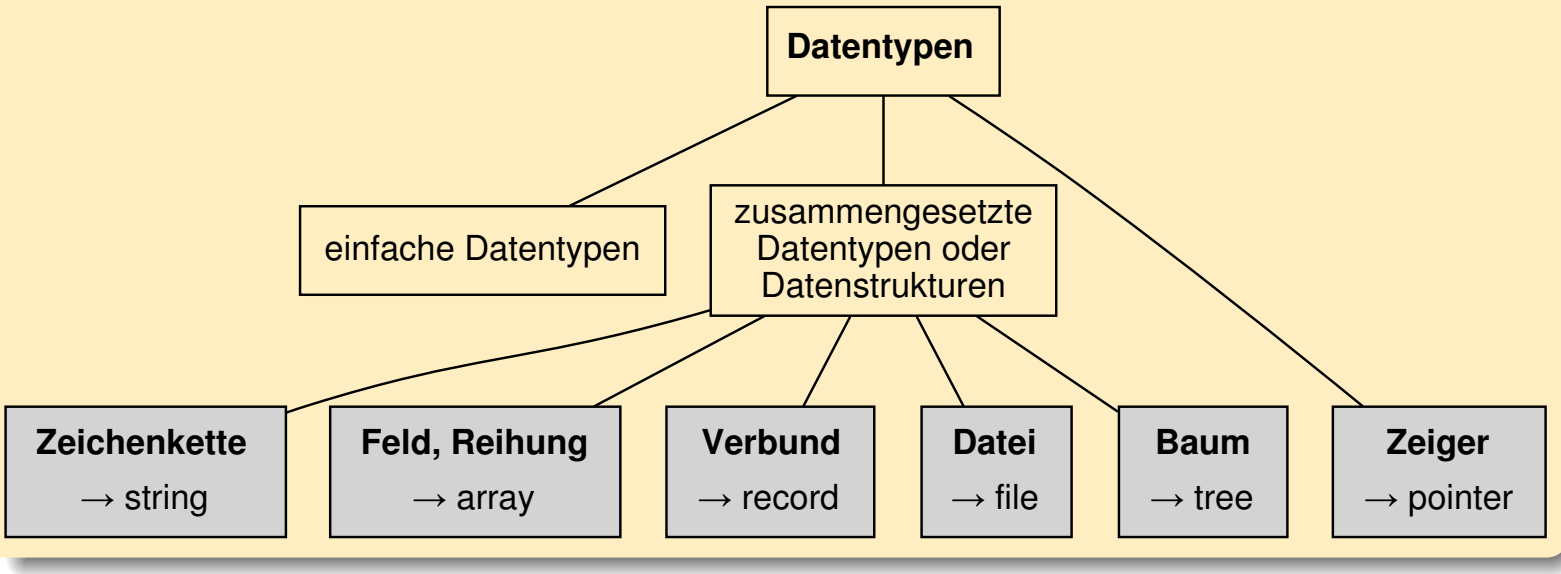
Datentypen und Datenstrukturen

Einteilung Teil 1: einfache Datentypen



Datentypen und Datenstrukturen

Einteilung Teil 2: zusammengesetzte Datentypen



einfache Datentypen

Zu den **einfachen Datentypen** zählen:

einfache Datentypen

Zu den **einfachen Datentypen** zählen:

- die ordinalen (zählbaren) Datentypen

einfache Datentypen

Zu den **einfachen Datentypen** zählen:

- die ordinalen (zählbaren) Datentypen
  - ganze Zahl (*integer*, *long integer* und weitere Bezeichnungen),

einfache Datentypen

Zu den **einfachen Datentypen** zählen:

- die ordinalen (zählbaren) Datentypen
  - ganze Zahl (*integer*, *long integer* und weitere Bezeichnungen),
  - Zeichen (character ➟ *char*) und

einfache Datentypen

Zu den **einfachen Datentypen** zählen:

- die ordinalen (zählbaren) Datentypen
  - ganze Zahl (*integer*, *long integer* und weitere Bezeichnungen),
  - Zeichen (character ➟ *char*) und
  - Wahrheits- oder logischer Wert (boolean ➟ *bool*).

einfache Datentypen

Zu den **einfachen Datentypen** zählen:

- die ordinalen (zählbaren) Datentypen
  - ganze Zahl (*integer*, *long integer* und weitere Bezeichnungen),
  - Zeichen (character ➟ *char*) und
  - Wahrheits- oder logischer Wert (boolean ➟ *bool*).

  Bei ihnen ist die Reihenfolge der Werte festgelegt. Jeder Wert außer dem letzten hat genau einen Nachfolger, und jeder Wert außer dem ersten hat genau einen Vorgänger.

einfache Datentypen

Zu den **einfachen Datentypen** zählen:

- die ordinalen (zählbaren) Datentypen
  - ganze Zahl (*integer*, *long integer* und weitere Bezeichnungen),
  - Zeichen (character ➠ *char*) und
  - Wahrheits- oder logischer Wert (boolean ➠ *bool*).

  Bei ihnen ist die Reihenfolge der Werte festgelegt. Jeder Wert außer dem letzten hat genau einen Nachfolger, und jeder Wert außer dem ersten hat genau einen Vorgänger.
- der nicht ordinale Datentyp der rationalen Zahl (Gleitkommazahl, *real* und weitere Bezeichnungen)

ganze Zahl

Wertebereiche:

meist 32 Bit $(-2^{31} \ldots 2^{31}-1)$, 16 Bit $(-2^{15} \ldots 2^{15}-1)$, 64 Bit $(-2^{63} \ldots 2^{63}-1)$

ganze Zahl

Wertebereiche:

meist 32 Bit $(-2^{31} \ldots 2^{31}-1)$, 16 Bit $(-2^{15} \ldots 2^{15}-1)$, 64 Bit $(-2^{63} \ldots 2^{63}-1)$

Bezeichnungen:

je nach Programmiersprache und Wertebereich z. B. *int*, *integer*, *long*, *longint*, *short*, *smallint*, *bigint*, ...

ganze Zahl

Wertebereiche:

meist 32 Bit $(-2^{31} \ldots 2^{31}-1)$, 16 Bit $(-2^{15} \ldots 2^{15}-1)$, 64 Bit $(-2^{63} \ldots 2^{63}-1)$

Bezeichnungen:

je nach Programmiersprache und Wertebereich z. B. *int*, *integer*, *long*, *longint*, *short*, *smallint*, *bigint*, ...

Operationen:

$+$, $-$, $*$, $<$, $>$, $=$, *div* (Division mit Rest) und *mod* (Modulo, Rest der Division)

ganze Zahl

Wertebereiche:
meist 32 Bit $(-2^{31} \ldots 2^{31}-1)$, 16 Bit $(-2^{15} \ldots 2^{15}-1)$, 64 Bit $(-2^{63} \ldots 2^{63}-1)$

Bezeichnungen:
je nach Programmiersprache und Wertebereich z. B. *int*, *integer*, *long*, *longint*, *short*, *smallint*, *bigint*, ...

Operationen:
$+$, $-$, $*$, $<$, $>$, $=$, *div* (Division mit Rest) und *mod* (Modulo, Rest der Division)

Bezeichnungen und Wertebereiche sind abhängig von der verwendeten Programmiersprache bzw. vom benutzten Anwendungsprogramm!

ganze Zahlen in ObjectPascal

Die verfügbaren Integertypen unterscheiden sich nach dem Speicherbedarf und ob ein Bit für das Vorzeichen verwendet wird oder nur positive (natürliche) Zahlen gespeichert werden können.

ganze Zahlen in ObjectPascal

Die verfügbaren Integertypen unterscheiden sich nach dem Speicherbedarf und ob ein Bit für das Vorzeichen verwendet wird oder nur positive (natürliche) Zahlen gespeichert werden können.

Außerdem gibt es Typen (sogenannte generische Typen – *Integer* und *Cardinal*), die abhängig vom verwendeten System (16 Bit, 32 Bit) unterschiedliche Wertebereiche haben können. Auf modernen Computern (32 oder 64 Bit-Systeme) spielt das allerdings keine Rolle.

ganze Zahlen in ObjectPascal

Die verfügbaren Integertypen unterscheiden sich nach dem Speicherbedarf und ob ein Bit für das Vorzeichen verwendet wird oder nur positive (natürliche) Zahlen gespeichert werden können.

Außerdem gibt es Typen (sogenannte generische Typen – *Integer* und *Cardinal*), die abhängig vom verwendeten System (16 Bit, 32 Bit) unterschiedliche Wertebereiche haben können. Auf modernen Computern (32 oder 64 Bit-Systeme) spielt das allerdings keine Rolle.

Die folgende Folie zeigt eine Übersicht der verfügbaren Typen mit Wertebereichen und Speicherbedarf.

ganze Zahlen in ObjectPascal

Typ	Wertebereich	Speicher
Shortint	$-128 \ldots 127$	1 Byte
Byte	$0 \ldots 255$	1 Byte
Smallint	$-32768 \ldots 32767$	2 Bytes
Word	$0 \ldots 65535$	2 Bytes
LongWord	$0 \ldots 42949672950$	4 Bytes
LongInt	$-2147483648 \ldots 2147483647$	4 Bytes
Int64	$-2^{63} \ldots 2^{63} - 1$	8 Bytes
Integer	$-2147483648 \ldots 2147483647$	4 Bytes
Cardinal	$0 \ldots 42949672950$	4 Bytes

ganze Zahlen in ObjectPascal

Implementation

ganze Zahlen in ObjectPascal

Implementation

Zur Variablendefinition dient das Schlüsselwort **`var`**. Lokale Ganzzahlen-Variablen werden innerhalb der Prozedur wie folgt definiert:

ganze Zahlen in ObjectPascal

Implementation

Zur Variablendefinition dient das Schlüsselwort **var**. Lokale Ganzzahlen-Variablen werden innerhalb der Prozedur wie folgt definiert:

```
procedure ...(Sender: TObject);
var a,b,c:integer;
    d:int64;
begin
// Implementationen
...
end;
```

ganze Zahlen in ObjectPascal

Werden globale Variablen benötigt (d. h. Variablen, die für das gesamte Programm gelten), erfolgt die Definition zwischen Deklaration und Implementation:

Quelltexteditor

Project1.lpr | *Unit1

```
    procedure Button1Click(Sender: TObject);
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;
  { hier weitere globale Variablen einfügen }
implementation

{$R *.lfm}

procedure TForm1.RadioButton1Click(Sender: TObject);
begin
  RadioButton2.Checked:=false;
```

55: 89 Geändert Einfg /home/wilke/Programmierung/Lazarus/Konvertierung/Unit1.pas

Aufgabe

1. Machen Sie sich mit der IDE Lazarus vertraut. Nutzen Sie dazu:
   - http://de.wikipedia.org/wiki/Lazarus_(Entwicklungsumgebung)
   - http://wiki.lazarus.freepascal.org/index.php/Main_Page/de
   - Die Präsentation „Programmieren mit Lazarus“
2. Programmieren Sie einen einfachen Taschenrechner für ganze Zahlen ($+$, $-$, $*$, *div* und *mod*).
3. Fangen Sie den möglichen Fehler „ganzzahlige Division durch Null“ unter Nutzung geeigneter Kontrollstrukturen ab.

weitere ordinale Typen in ObjektPascal

Zeichen

weitere ordinale Typen in ObjektPascal

Zeichen

Der Datentyp *Char* steht für einzelne Zeichen. Die Kodierung ist abhängig vom Betriebssystem (Umlaute etc.). Der Speicherbedarf beträgt ein Byte (8 Bit, Wertebereich: $0 \ldots 255$).

weitere ordinale Typen in ObjektPascal

Zeichen

Der Datentyp *Char* steht für einzelne Zeichen. Die Kodierung ist abhängig vom Betriebssystem (Umlaute etc.). Der Speicherbedarf beträgt ein Byte (8 Bit, Wertebereich: $0 \ldots 255$).

Char eignet sich **nicht** zum rechnen!

weitere ordinale Typen in ObjektPascal

Zeichen

Der Datentyp *Char* steht für einzelne Zeichen. Die Kodierung ist abhängig vom Betriebssystem (Umlaute etc.). Der Speicherbedarf beträgt ein Byte (8 Bit, Wertebereich: 0 . . . 255).

Char eignet sich **nicht** zum rechnen!

Zuweisung gültiger Werte:

```
var f,g:char;
...
f := 'A';
g := '#64'; // ASCII-Code 64 = '@'
```

weitere ordinale Typen in ObjektPascal

Wahrheitswerte

weitere ordinale Typen in ObjektPascal

Wahrheitswerte

Wertebereich: true . . . false

weitere ordinale Typen in ObjektPascal

Wahrheitswerte

Wertebereich: true . . . false

Speicherbedarf: 1 Byte

weitere ordinale Typen in ObjektPascal

Wahrheitswerte

Wertebereich: true . . . false

Speicherbedarf: 1 Byte

Definition und Zuweisung:

```
var d,e:boolean;
...
d := true;
e := false;
```

weitere ordinale Typen in ObjektPascal

Wahrheitswerte

Wertebereich: true . . . false

Speicherbedarf: 1 Byte

Definition und Zuweisung:

```
var d,e:boolean;
...
d := true;
e := false;
```

Weitere Datentypen siehe z. B.:
http://wiki.freepascal.org/Data_type/de

Fließkommatypen (rationale Zahlen)

Übersicht:

Typ	Bereich	Genauigk*	Speich.
Single	$1,5 \cdot 10^{-45} \ldots 3,4 \cdot 10^{38}$	7 – 8 St.	4 Bytes
Double	$5 \cdot 10^{-324} \ldots 1,7 \cdot 10^{308}$	15 – 16 St.	8 Bytes
Extended	Plattformabhängig, bei Intel 80x86-Prozessoren:		
	$3,4 \cdot 10^{-4932} \ldots 1,1 \cdot 10^{4932}$	19 – 20 St.	10 Bytes
Comp	$-9,2 \cdot 10^{18} \ldots 9,2 \cdot 10^{18}$	19 – 20 St.	8 Bytes
Real	abhängig vom verwendeten Prozessor, bei aktuellen Prozessoren wie Double		

* Genauigkeit als Anzahl der Kommastellen

Weitere Fließkommatypen siehe z. B.:
http://wiki.freepascal.org/Data_type/de

Fließkommatypen (rationale Zahlen)

Definition und Zuweisung:

```
var a,b:real;
...
a := -123.45678; // Punkt statt Komma!
b := 9876.54321;
```

Fließkommatypen (rationale Zahlen)

Definition und Zuweisung:

```
var a,b:real;
...
a := -123.45678; // Punkt statt Komma!
b := 9876.54321;
```

Formatumwandlungen (Beispiele):

- Text zu Zahl:

  ```
  a := strtofloat(edit1.text);
  ```

- Zahl zu Text:

  ```
  edit3.text := floattostr(c);
  ```

Aufgaben

1. Modifizieren Sie den einfachen Taschenrechner aus dem ersten Aufgabenkomplex für Fließkommazahlen. (➠ Formatumwandlungen anpassen!)
2. Lagern Sie die Ausgabeoperationen in eine eigene Prozedur aus *(siehe nächste Folie)*.
3. Sorgen Sie dafür, dass negative Ergebnisse rot angezeigt werden. Verwenden Sie dazu den Datentyp *Boolean* und eine geeignete Kontrollstruktur.

   Um die Textfarbe einer Edit-Komponente zu ändern, benutzen Sie z. B. folgende Zuweisung:

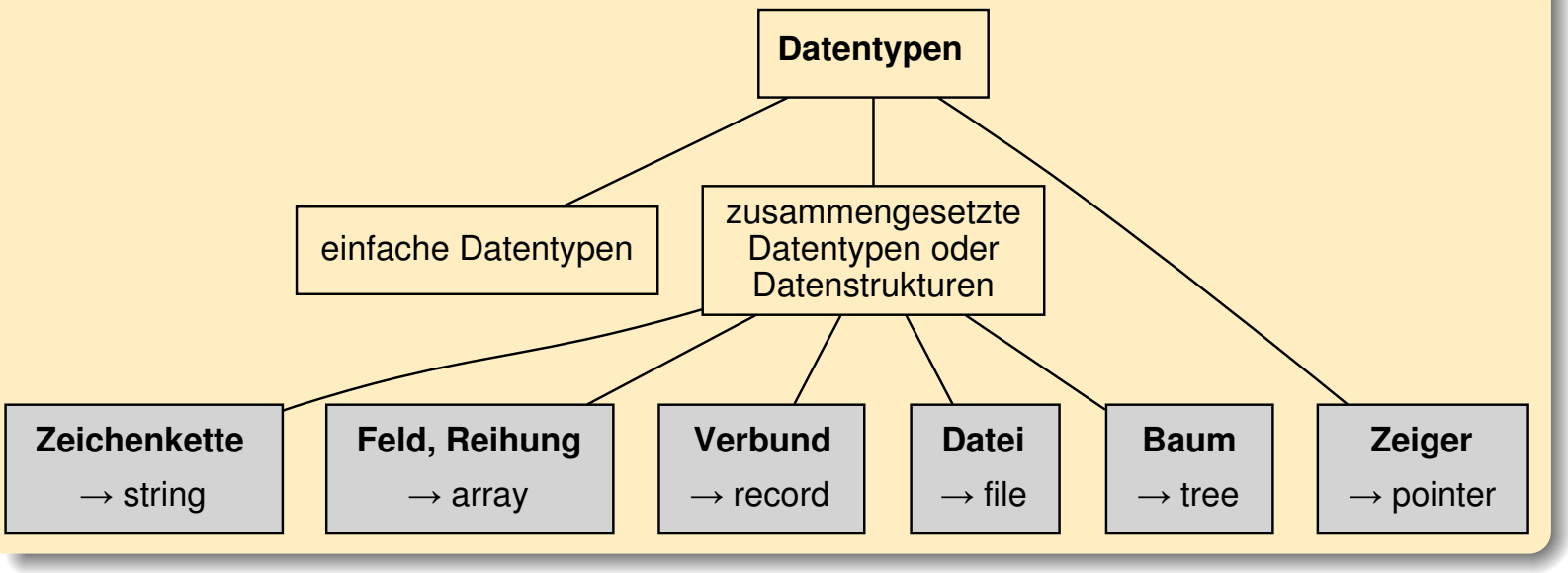
   ```
   edit1.Font.Color := clred;
   ```

Hinweis zur Aufgabe 2

1. Variablen (falls nicht schon geschehen) global definieren.
2. Prozedur für Ausgabe erzeugen:
   ```
   procedure TForm1.Ausgabe;
   begin
   ...
   end;
   ```
3. Prozedurkopf (`prozedure Ausgabe;`) im Deklarationsteil bei `TForm1` einfügen.
4. Aufruf der Prozedur `Ausgabe` an allen Stellen, wo eine Ausgabe erfolgen soll (z. B. jeweils am Ende der Berechnungen).

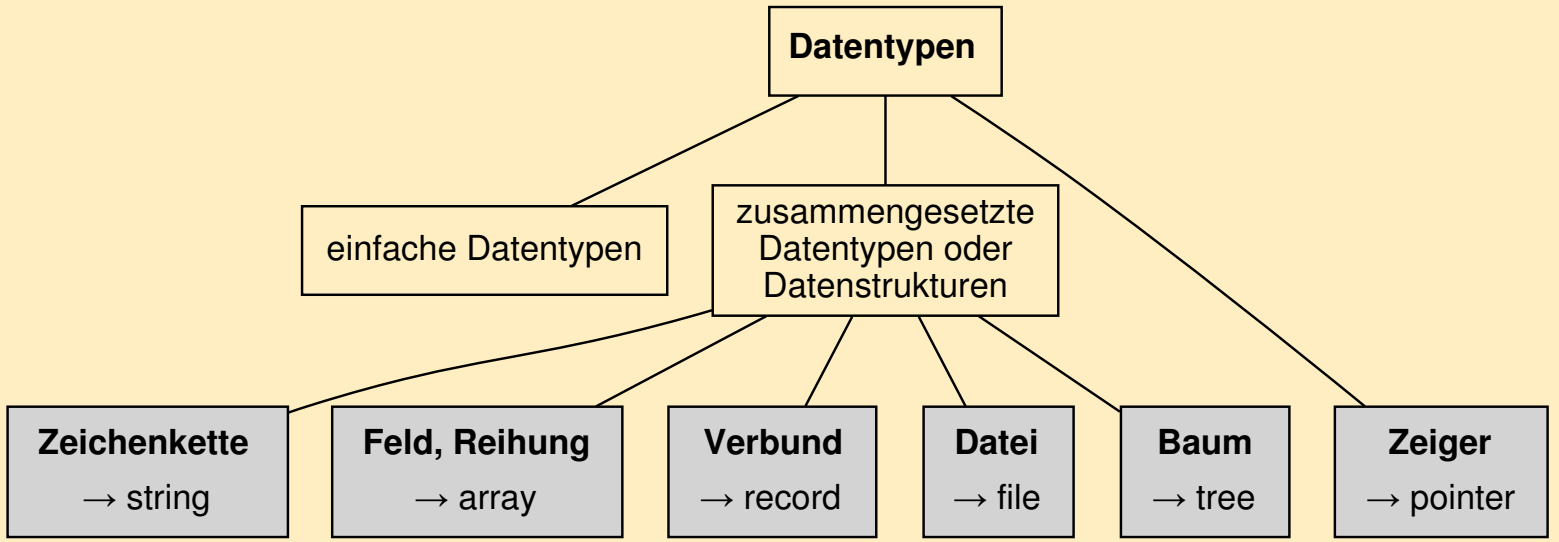
zusammengesetzte Datentypen

Wiederholung: zusammengesetzte Datentypen



zusammengesetzte Datentypen

Wiederholung: zusammengesetzte Datentypen



Am Beispiel ObjektPascal wird auf den folgenden Folien die Benutzung der Datenstrukturen *string*, *array* und *record* gezeigt. Datei und Baum werden nur theoretisch vorgestellt.

Die Zeichenkette *(string)*

Grundlagen

Der Datentyp *string* steht für eine Zeichenkette, deren Eigenschaften von einem Compilerswitch abhängen.

Die Zeichenkette *(string)*

Grundlagen

Der Datentyp *string* steht für eine Zeichenkette, deren Eigenschaften von einem Compilerswitch abhängen.

Das Datenfeld vom Datentyp *string* ist ein Feld *(array)*, der aus Datenfeldern des Datentyps *char* besteht.

Die Zeichenkette *(string)*

Grundlagen

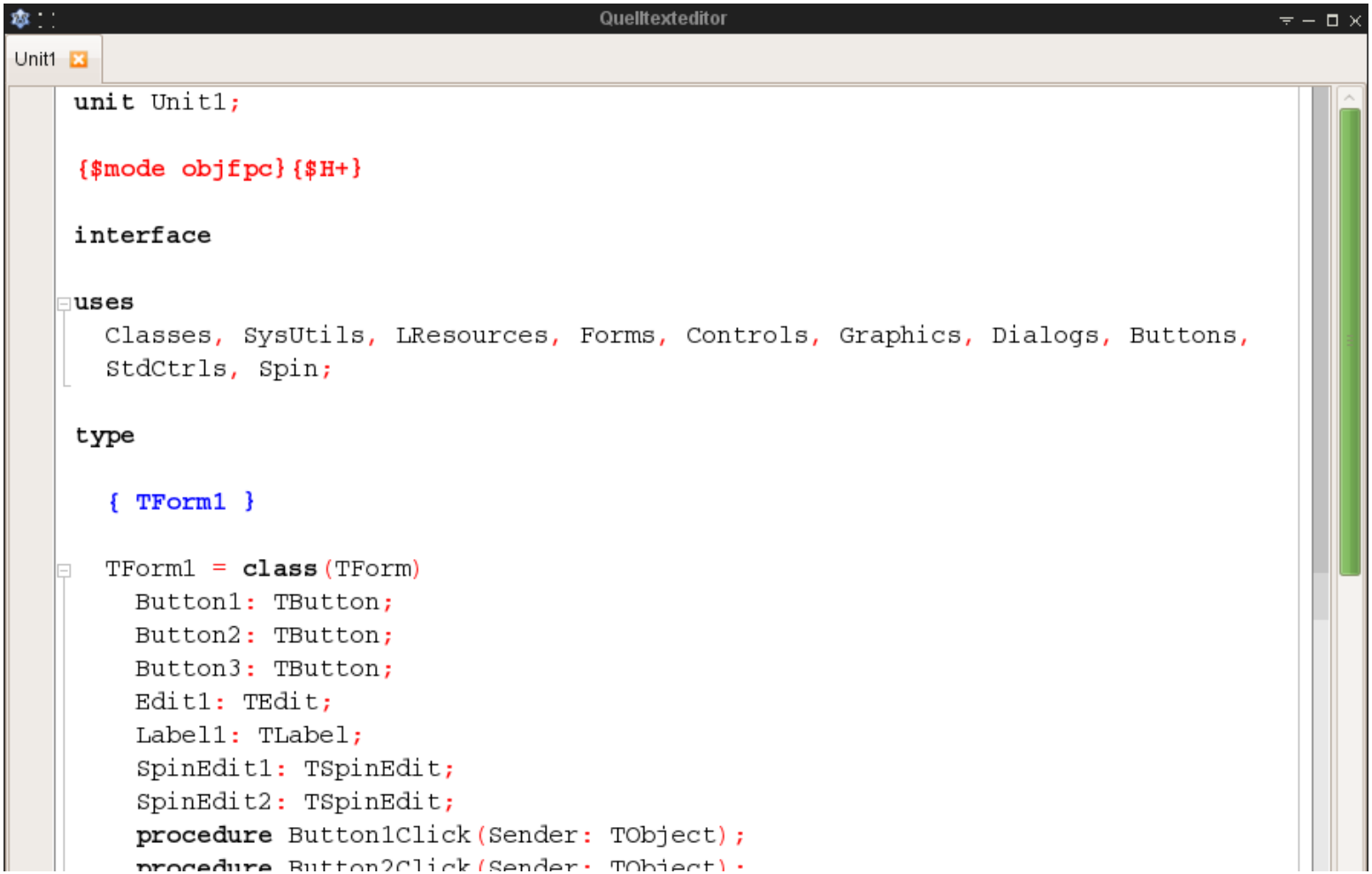
Der Datentyp *string* steht für eine Zeichenkette, deren Eigenschaften von einem Compilerswitch abhängen.

Das Datenfeld vom Datentyp *string* ist ein Feld *(array)*, der aus Datenfeldern des Datentyps *char* besteht.

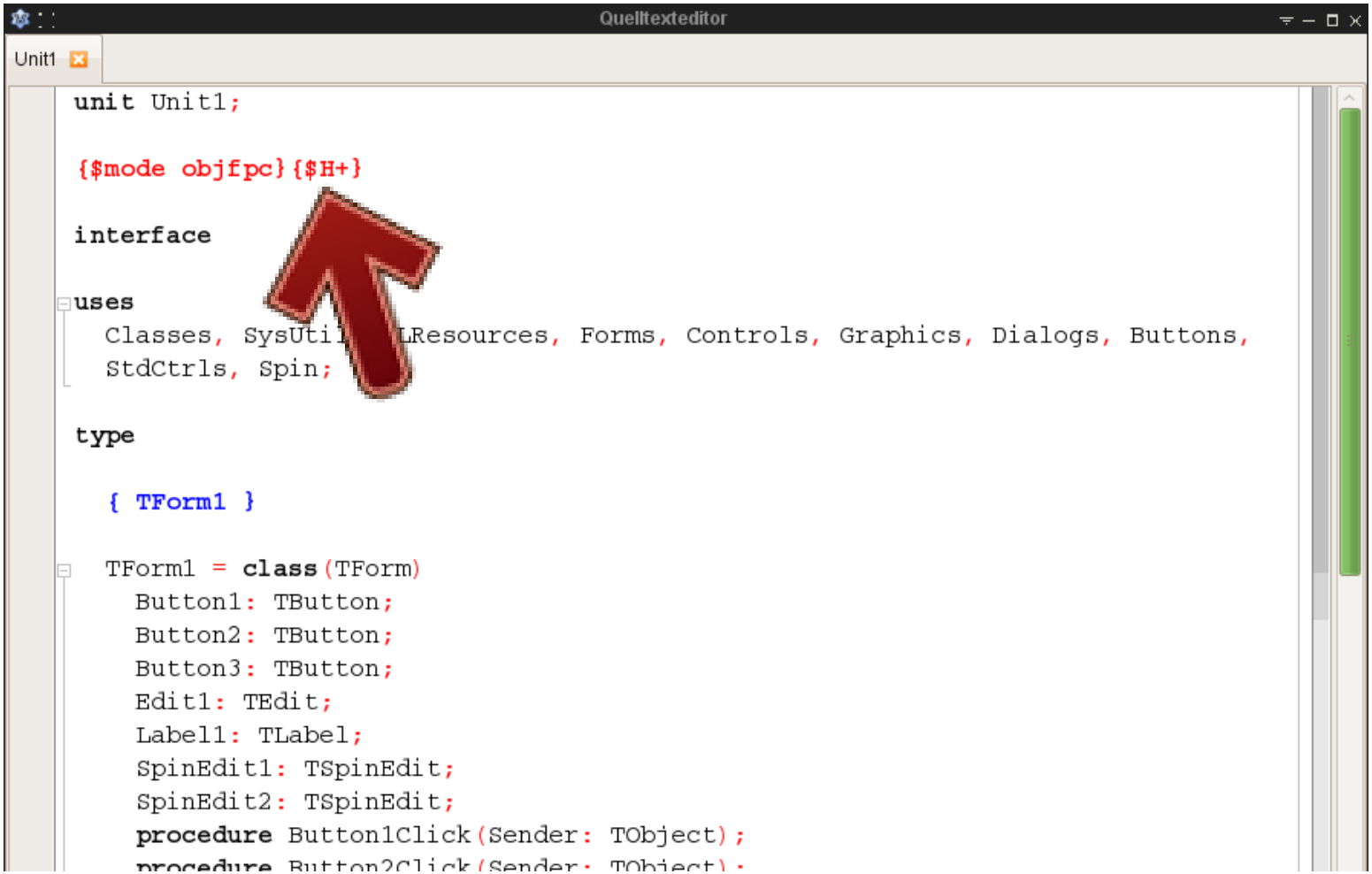
Die maximale Länge eines *strings* wird durch einen sogenannten Compilerswitch im Kopf der unit bestimmt:

- `{$H-}` ➠ maximale Länge von 255 Byte (entspricht dem Datentyp *ShortString*)
- `{$H+}` ➠ unbegrenzte Länge (entspricht dem Datentyp *AnsiString*)

Compilerswitch



Compilerswitch



Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

```
var s:string;
```

Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

```
var s:string;
```

Zuweisung gültiger Werte:

```
s := 'Hallo Welt!';
s := '42';
s := inttostr(42);
```

Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

```
var s:string;
```

Zuweisung gültiger Werte:

```
s := 'Hallo Welt!';
s := '42';
s := inttostr(42);
```

Stringmanipulationen:

- Verketten: `s := s + ' ist die Lösung.';`

Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

```
var s:string;
```

Zuweisung gültiger Werte:

```
s := 'Hallo Welt!';
s := '42';
s := inttostr(42);
```

Stringmanipulationen:

- Verketten: `s := s + ' ist die Lösung.';`
- Länge ermitteln: `slaenge := Length(s);`

Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

```
var s:string;
```

Zuweisung gültiger Werte:

```
s := 'Hallo Welt!';
s := '42';
s := inttostr(42);
```

Stringmanipulationen:

- Verketten: `s := s + ' ist die Lösung.';`
- Länge ermitteln: `slaenge := Length(s);`
- Zugriff auf Position: `zeichen_5 := s[5];`

Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

```
var s:string;
```

Zuweisung gültiger Werte:

```
s := 'Hallo Welt!';
s := '42';
s := inttostr(42);
```

Datentyp *integer*

Stringmanipulationen:

- Verketten: `s := s + ' ist die Lösung.';`
- Länge ermitteln: `slaenge := Length(s);`
- Zugriff auf Position: `zeichen_5 := s[5];`

Die Zeichenkette *(string)*

Umgang mit Zeichenketten

Variablendefinition:

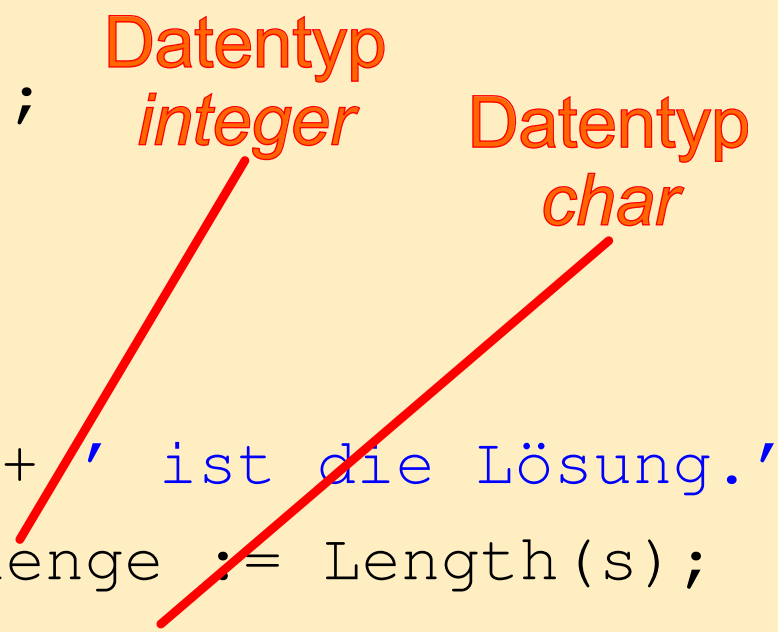
```
var s:string;
```

Zuweisung gültiger Werte:

```
s := 'Hallo Welt!';
s := '42';
s := inttostr(42);
```

Stringmanipulationen:

- Verketten: `s := s + ' ist die Lösung.';`
- Länge ermitteln: `slaenge := Length(s);`
- Zugriff auf Position: `zeichen_5 := s[5];`



Das Feld *(array)*

Grundlagen

Ein *array* ist eine Gruppe von Datenfeldern des selben Datentyps.

Das Feld *(array)*

Grundlagen

Ein *array* ist eine Gruppe von Datenfeldern des selben Datentyps.

Ein *array* kann aus allen einfachen Datentypen gebildet werden. (Außerdem sind *arrays* aus *records* und Klassen möglich.)

Das Feld *(array)*

Grundlagen

Ein *array* ist eine Gruppe von Datenfeldern des selben Datentyps.

Ein *array* kann aus allen einfachen Datentypen gebildet werden. (Außerdem sind *arrays* aus *records* und Klassen möglich.)

Man unterscheidet:

- eindimensionale und mehrdimensionale *arrays*
- statische *arrays* mit fester Größe und dynamische *arrays* mit veränderlicher Größe

Das Feld *(array)*

Grundlagen

Ein *array* ist eine Gruppe von Datenfeldern des selben Datentyps.

Ein *array* kann aus allen einfachen Datentypen gebildet werden. (Außerdem sind *arrays* aus *records* und Klassen möglich.)

Man unterscheidet:

- eindimensionale und mehrdimensionale *arrays*
- statische *arrays* mit fester Größe und dynamische *arrays* mit veränderlicher Größe

Der Datentyp *string* ist z. B. ein eindimensionaler dynamischer (bei Compilerswitch `{$H+}`) *array* des Datentyps *char*.

Das Feld *(array)*

statisches eindimensionales *array*

Variablendefinition:

```
var
  arrint1: array[0..2] of integer;
       // enthält drei Elemente
  arrint2: array[4..6] of integer;
       // enthält auch drei Elemente
```

Das Feld *(array)*

statisches eindimensionales *array*

Variablendefinition:

```
var
  arrint1: array[0..2] of integer;
       // enthält drei Elemente
  arrint2: array[4..6] of integer;
       // enthält auch drei Elemente
```

Grafische Darstellung der beiden *arrays*:

arrint1:	0	1	2

arrint2:	4	5	6

Das Feld *(array)*

statisches eindimensionales *array* (Fortsetzung)

Zugriff auf Daten im *array*:
Ein Element wird mit dem Namen des *arrays* und der Position angesprochen.

Das Feld *(array)*

statisches eindimensionales *array* (Fortsetzung)

Zugriff auf Daten im *array*:
Ein Element wird mit dem Namen des *arrays* und der Position angesprochen.

Für arrint1 (0 | 1 | 2) liefert z. B. `arrint1[1]` das zweite Element mit der Position 1 (0 | **1** | 2) zurück.

Das Feld *(array)*

dynamisches eindimensionales *array*

Variablendefinition:

```
var
  arrint: array of integer;
      // enthält null Elemente
```

Das Feld *(array)*

dynamisches eindimensionales *array*

Variablendefinition:

```
var
  arrint: array of integer;
      // enthält null Elemente
```

Festlegung der Größe des *arrays*:

```
SetLength(arrint, 20);
   // Größe wird auf 20 El. geändert
```

Das Feld *(array)*

dynamisches eindimensionales *array*

Variablendefinition:

```
var
  arrint: array of integer;
      // enthält null Elemente
```

Festlegung der Größe des *arrays*:

```
SetLength(arrint, 20);
   // Größe wird auf 20 El. geändert
```

Praxisbeispiel: `arrint` auf Länge des Strings `s1` ändern:

```
SetLength(arrint, Length(s1));
```

Aufgaben

1. Informieren Sie sich über mehrdimensionale *arrays*.
   - ➠ http://wiki.freepascal.org/Array/de
2. Planen (evtl. Gruppenarbeit) und implementieren Sie ein objektorientiertes Programm zur Vigenère-Verschlüsselung. Zur Orientierung:
   - globale Variablendefinitionen für Klartext, Schlüsseltext, Schlüssel (*strings*) und dynamische *arrays* vom Typ *integer* für die Berechnungen
   - Prozeduren für Einlesen, Eingabe*strings* auf *arrays* vom Typ *integer* abbilden (case, for-Schleife, . . .), Verschlüsseln (Addition), Entschlüsseln (Subtraktion), Ergebnisse wieder auf *string* abbilden und Ausgabe

Der Verbund *(record)*

Grundlagen

Ein *record* ist ein hoch strukturierter und komplexer Datentyp. Man bezeichnet ihn auch als **Datensatz.**

Der Verbund *(record)*

Grundlagen

Ein *record* ist ein hoch strukturierter und komplexer Datentyp. Man bezeichnet ihn auch als **Datensatz.**

Im Gegensatz zum *array* kann ein *record* aus Elementen (➠ Feldern) unterschiedlicher Typen bestehen.

Der Verbund *(record)*

Grundlagen

Ein *record* ist ein hoch strukturierter und komplexer Datentyp. Man bezeichnet ihn auch als **Datensatz.**

Im Gegensatz zum *array* kann ein *record* aus Elementen (➠ Feldern) unterschiedlicher Typen bestehen.

Records werden zunächst im Kopf der unit (➠ unter `type`) definiert, bevor Variablen vom definierten Typ erzeugt werden können.

Der Verbund *(record)*

Grundlagen

Ein *record* ist ein hoch strukturierter und komplexer Datentyp. Man bezeichnet ihn auch als **Datensatz.**

Im Gegensatz zum *array* kann ein *record* aus Elementen (➠ Feldern) unterschiedlicher Typen bestehen.

Records werden zunächst im Kopf der unit (➠ unter `type`) definiert, bevor Variablen vom definierten Typ erzeugt werden können.

Das folgende Beispiel zeigt die Verwendung eines *records* am Beispiel einer kleinen Personaldatenbank.

Der Verbund *(record)*

❶ Die Typdefinition

```
type
   Mitarbeiter = record
      Vorname, Nachname: string;
      Adresse: array[1..3] of string;
      Telefon: integer;
      Geburtsdatum: TDateTime;
      Lohn_erhalten: boolean;
   end;
```

Eigene Typdefinitionen gehören in den private-Bereich, wenn nur die entsprechende unit darauf zugreifen soll. Sollen weitere units Zugriff erhalten, nutzt man den public-Bereich.

Der Verbund *(record)*

❶ Die Typdefinition

Quelltexteditor

Unit1

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

end.
```

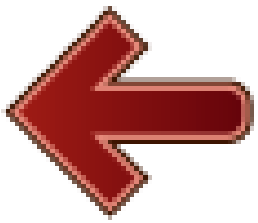
Der Verbund *(record)*

❶ Die Typdefinition

Quelltexteditor

Unit1

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

end.
```



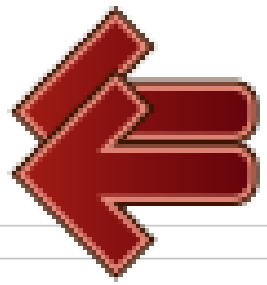
Der Verbund *(record)*

❶ Die Typdefinition

Quelltexteditor

Unit1

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs;

type
  TForm1 = class(TForm)
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

end.
```



Der Verbund *(record)*

❷ Die Variablendefinition

```
var a, b, c, d: Mitarbeiter;
```

Der Verbund *(record)*

❷ Die Variablendefinition

```
var a, b, c, d: Mitarbeiter;
```

❸ Zugriff auf die Variablen (a) mittels Punktoperator

```
a.Vorname := 'Heinz';
a.Telefon := 5550123456789;
```

Der Verbund *(record)*

❷ Die Variablendefinition

```
var a, b, c, d: Mitarbeiter;
```

❸ Zugriff auf die Variablen (a) mittels Punktoperator

```
a.Vorname := 'Heinz';
a.Telefon := 5550123456789;
```

❸ Zugriff auf die Variablen (b) mittels WITH-Konstrukt

```
with a do
begin
   Vorname := 'Heinz';
   Nachname := 'Schmidt';
...
end;
```

Die Datei *(file)*

- Dateien sind Zusammenfassungen von Daten gleicher Struktur. Jede in der Programmiersprache vereinbarte Datenstruktur – außer *file* selbst – kann gespeichert werden.

Die Datei *(file)*

- Dateien sind Zusammenfassungen von Daten gleicher Struktur. Jede in der Programmiersprache vereinbarte Datenstruktur – außer *file* selbst – kann gespeichert werden.
- Kleine Dateien werden aufeinanderfolgend (sequenziell) gespeichert. Größere Dateien werden anders organisiert (z. B. in Baumstrukturen).

Die Datei *(file)*

- Dateien sind Zusammenfassungen von Daten gleicher Struktur. Jede in der Programmiersprache vereinbarte Datenstruktur – außer *file* selbst – kann gespeichert werden.
- Kleine Dateien werden aufeinanderfolgend (sequenziell) gespeichert. Größere Dateien werden anders organisiert (z. B. in Baumstrukturen).
- Dateien werden unter einem Dateinamen auf dem Datenträger gespeichert. Sie können ständig erweitert werden (➠ dynamische Datenstruktur).

Der Baum *(tree)*

- Bäume enthalten Daten (➠ Knoten, *nodes*) in einer hierarchischen Anordnung:

Der Baum *(tree)*

- Bäume enthalten Daten (➠ Knoten, *nodes*) in einer hierarchischen Anordnung:
  - Jeder Knoten – außer der Wurzel – hat genau einen Vorgänger.

Der Baum *(tree)*

- Bäume enthalten Daten (➠ Knoten, *nodes*) in einer hierarchischen Anordnung:
  - Jeder Knoten – außer der Wurzel – hat genau einen Vorgänger.
  - Jeder Knoten – außer Endknoten – hat mehrere Nachfolger.

Der Baum *(tree)*

- Bäume enthalten Daten (➠ Knoten, *nodes*) in einer hierarchischen Anordnung:
  - Jeder Knoten – außer der Wurzel – hat genau einen Vorgänger.
  - Jeder Knoten – außer Endknoten – hat mehrere Nachfolger.

Aufgaben – schriftlich, Hausaufgabe

1. Informieren sie sich über Baumstrukturen aus Sicht der Graphentheorie. *(nur grober Überblick!)*
2. Stellen Sie Anwendungsbeispiele für Baumstrukturen in der Informatik zusammen.