

Einführung in die Programmierung

weiterführende Konzepte: Pointer, Funktionen, Parameterübergabe

Prof. Dr. Peter Jüttner

Pointer

1. Pointer

- **Pointer (auch Zeiger genannt), zeigt auf eine Stelle im Speicher, ausgedrückt durch eine (physikalische) Speicheradresse**
- **Pointer kann eine Konstante (eher selten, Ausnahmen Register, NULL-Pointer) oder Variable sein**
- **Inhalt einer Variable vom Typ Pointer: Speicheradresse**
- **Pointervariable steht wie alle Variablen selbst im Speicher**



Pointer

1. Pointer

Pointer

**Pointer-
variable**

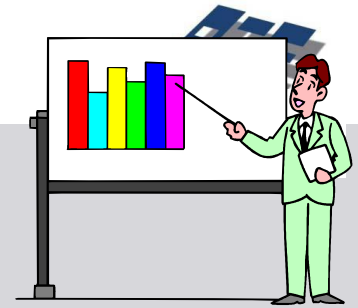
**Adresse der
Pointer-
variablen**

Adr.	Speicher*)
1	
2	
3	7153
4	
5	
...	
7153	„a“
...	
9999	
10000	

*) Speicher kann ein beliebiger Speicher sein, z. B. RAM, ROM, Register



Pointer



1. Pointer

- sind (meistens) typisiert, d.h. zeigen auf Speicherinhalt eines bestimmten Typs (mit Größe und Struktur), z.B. Pointer auf Integer, Pointer auf Float, Pointer auf eine Struktur
 - ➔ der Inhalt des Speichers, auf den der Pointer zeigt, kann entsprechend dem Typ behandelt werden (Operationen, Parameter)
 - ➔ Pointer verschiedener Typen dürfen nicht „vermischt“ werden (Zuweisungen, Zugriffe)

Pointer

1. Pointer

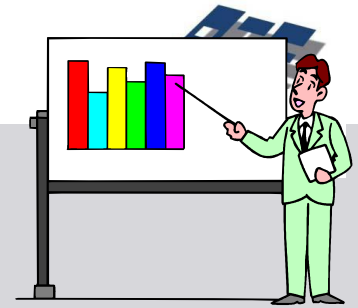
**Pointer-
variable
auf komplexe Zahl
mit Real- und
Imaginärteil**

Adr.	Speicher*)
1	
2	
3	7153
4	
5	
...	
7153	float real
7154	float imag
...	
10000	



*) Speicher kann ein beliebiger Speicher sein, z. B. RAM, ROM, Register

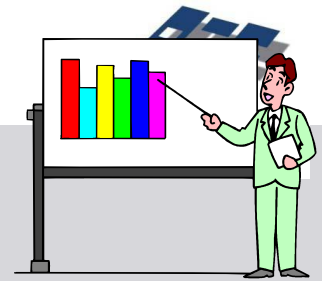
Pointer



1. Pointer

- **Pointer haben auf einer HW alle die gleiche Größe (z.B. 4 Bytes)**
- **Pointer können auch auf Dateien (s. File In/Output) oder Funktionen zeigen**
- **Pointer werden auch als Referenz bezeichnet**

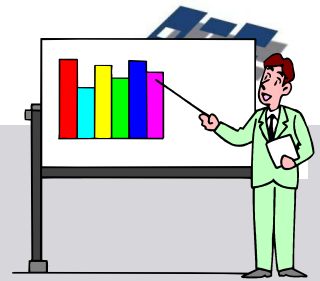
Pointer



1. Pointer

- **Verwendung in dynamischen Datenstrukturen (Listen, Bäume)**
- **Verwendung in der dynamischen Speicherverwaltung**
- **Verwendung in HW-naher Programmierung, Ansprechen von Registern, Ports, Interrupttabellen**
- **Verwendung zur Parameterübergabe**
- **Verwendung zur Resultatübergabe**

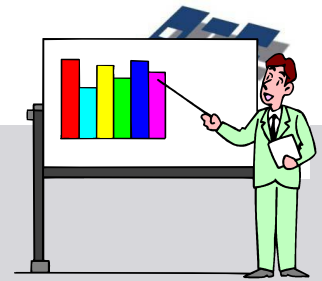
Pointer



1. Pointer

- Deklaration Pointervariable in C: *Typ* Pointername*
- * drückt aus, dass es sich um einen Pointer handelt z.B.
 - `int* intpointer; /* Pointer auf einen Integer */`
 - `char* charpointer /* Pointer auf Character */`
 - `int* register_X /* Pointer auf ein Register */`
 - `Struktur* structpointer /* Pointer auf Datenstruktur */`
 - `void* p /* „reine“ Speicheradresse, keine Typisierung */`

Pointer



1. Pointer

- **Pointer als Ergebnis einer Funktion in C:**
typ f(typ1 p1, ...)*
- *** drückt aus, dass die Funktion einen Pointer (also eine Speicheradresse) zurückgibt.**
- **Das eigentliche Ergebnis der Funktion steht meist an der zurückgegebenen Speicherstelle**

Pointer

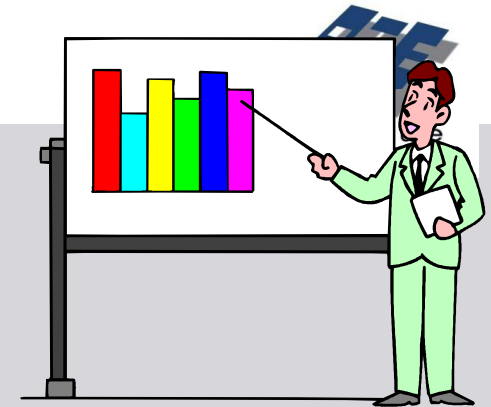
1. Pointer

**Pointer als
Ergebnis einer
Funktion**

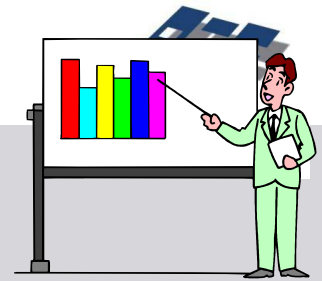
**Inhalt wird weiter-
verarbeitet**

*) Speicher kann ein beliebiger
Speicher sein, z. B. RAM, ROM,
Register

Adr.	Speicher*)
1	
2	
3	
4	
5	
...	
7153	„a“
...	
9999	
10000	



Pointer



1. Pointer

- **Pointer als Parameter einer Funktion in C:**
ergtyp $f(typ^* p, \dots)$
- *** drückt aus, dass die Funktion einen Pointer (also eine Speicheradresse) als Parameter hat.**
- **Der eigentliche Parameter der Funktion steht meist an der übergebenen Speicherstelle**

Pointer

1. Pointer

**Pointer als
Parameter einer
Funktion**

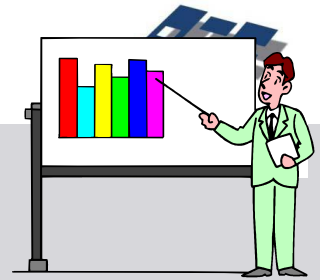
**Funktion greift
i.d.R. auf
Inhalt zu**

*) Speicher kann ein beliebiger Speicher sein, z. B. RAM, ROM, Register

Adr.	Speicher*)
1	
2	
3	
4	
5	
...	
7153	„a“
...	
9999	
10000	



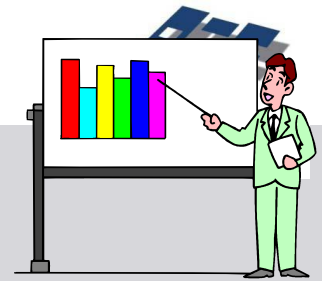
Pointer



1. Pointer

- **Pointer als Parameter und / oder Ergebnis einer Funktion**
 - **erspart Kopieren großer Datenmengen auf Parameter- oder Ergebnisposition**
 - **ist bei dynamischen Strukturen ohne Alternative**
 - **erfordert Vorsicht bei der Anwendung, da der Speicher, auf den der Pointer zeigt i.d.R. verändert wird**

Pointer

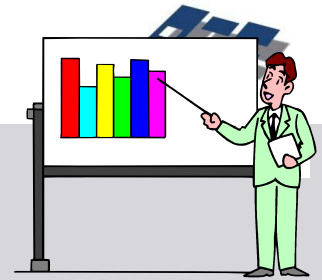


1. Pointer

- **Pointervariable, Belegung mit einem Wert**
 - **`intpointer = NULL; /* NULLPointer, zeigt nirgendwohin */`**
 - **`charpointer = 0xFF01; /* feste Adresse */`**
 - **`register_X = 0xFFAA /* feste Adresse */`**
 - **`Struktur *structpointer = &s; /* Adresse einer Variablen im RAM */`**
 - **`pointer1 = pointer2; /* Wert eines anderen Pointers , beide zeigen auf gleichen Typ */`**
 - **`intpointer = charpointer; /* Verboten! */`**



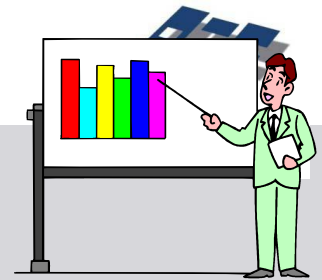
Pointer



1. Pointer

- **Pointervariable, Dereferenzieren**
 - **Zugriff auf den Speicherinhalts, auf den der Pointer zeigt**
 - **liefert lesend einen Wert von Typ auf den der Pointer zeigt**
 - **liefert schreibend eine Speicherstelle von Typ auf den der Pointer zeigt**
 - **dieser Wert oder die Speicherstelle können in Operationen oder als Parameter oder als Ergebnis einer Funktion weiterverarbeitet werden**

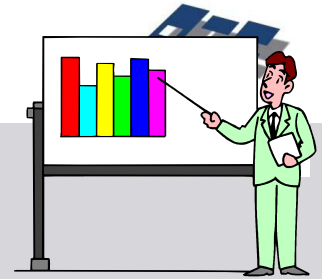
Pointer



1. Pointer

- **Pointervariable, Dereferenzieren**
 - **Dereferenzieren in C: *pointername wird in einem Ausdruck verwendet, wo der Typ des Pointers verwendet werden darf**

Pointer



1. Pointer

- **Pointervariable, Dereferenzieren**

**Pointer (lesend)
dereferenziert
liefert Wert von
Adresse
(„schaut dort nach
wo der Pointer hinzeigt“)**

Adr.	Speicher*)
1	
2	
3	
4	7153
5	
...	
7153	Wert
...	
9999	
10000	

Diagram illustrating pointer dereferencing:

- A red arrow points from the text "Pointer (lesend) dereferenziert liefert Wert von Adresse" to the address 7153 in the "Adr." column.
- A red arrow points from the text "dereferenziert" to the value "Wert" in the "Speicher*" column at address 7153.
- A red arrow points from the text "liefert Wert von Adresse" to the value "Wert" in the "Speicher*" column at address 7153.
- A red arrow points from the text "(„schaut dort nach wo der Pointer hinzeigt“)" to the value "Wert" in the "Speicher*" column at address 7153.
- A red arrow points from the text "Wert" (on the left) to the value "Wert" in the "Speicher*" column at address 7153.

Pointer



1. Pointer

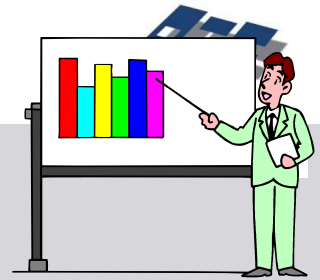
- **Pointervariable, Dereferenzieren**

**Pointer (schreibend)
dereferenziert
schreibt Wert
an Adresse
(„schaut dort nach
wo der Pointer hinzeigt“)**

Wert

Adr.	Speicher*)
1	
2	
3	
4	7153
5	
...	
7153	
...	
9999	
10000	

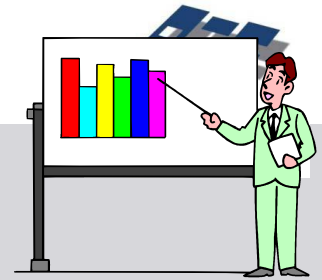
Pointer



1. Pointer

- **Pointervariable, Dereferenzieren**
 - `*charpointer = 'c';`
 - `*register_X = 0xAA /* Register beschreiben */`
 - `f1 (*structpointer) /* Parameter */`
 - `x = *intpointer1 + *intpointer2; /* Addition der Werte, auf die intpointer1 und intpointer2 zeigen */`
 - `... return *intpointer; /* Zurückgeben eines Funktionsergebnisses */`

Pointer



1. Pointer

- Pointervariable, Dereferenzieren
- Ein Pointer, der dereferenziert werden soll, muss immer auf eine definierte Speicheradresse zeigen!

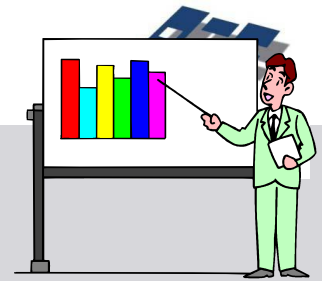


- `char* charpointer = NULL;`
`*charpointer = 'a'; /* Verboten! */`



- `char *charpointer;`
`char c;`
`c = *charpointer; /* Verboten! */`

Pointer



1. Pointer

- **Pointervariable, Dereferenzieren**

- **Pointer auf Strukturen**

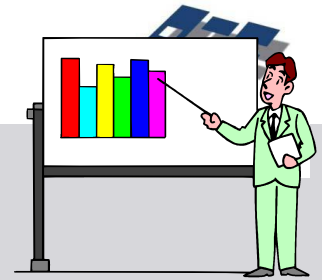
```
typedef struct complex /* Struktur für komplexe Zahl */  
{ float real;  
  float imag;  
};  
complex *cpointer;
```

- **2 Möglichkeiten des Komponentenzugriffs bei Dereferenzierung**

1. **`(*cpointer).real = 5.5;`**

2. **`cpointer->real = 5.5;`**

Pointer



2. Pointer und Arrays

- `int intfeld[10];` definiert ein `int` Feld mit 10 Elementen
- `intfeld` ist der Name des Felds kann in C aber auch als Adresse (i.e. Pointer) auf das 1. Element (index 0) betrachtet werden.
- `intfeld` kann an Pointervariable von Typ `int*` zugewiesen werden
- `int* ipointer = intfeld`
- Arrays werden nur als Pointer an Funktionen übergeben

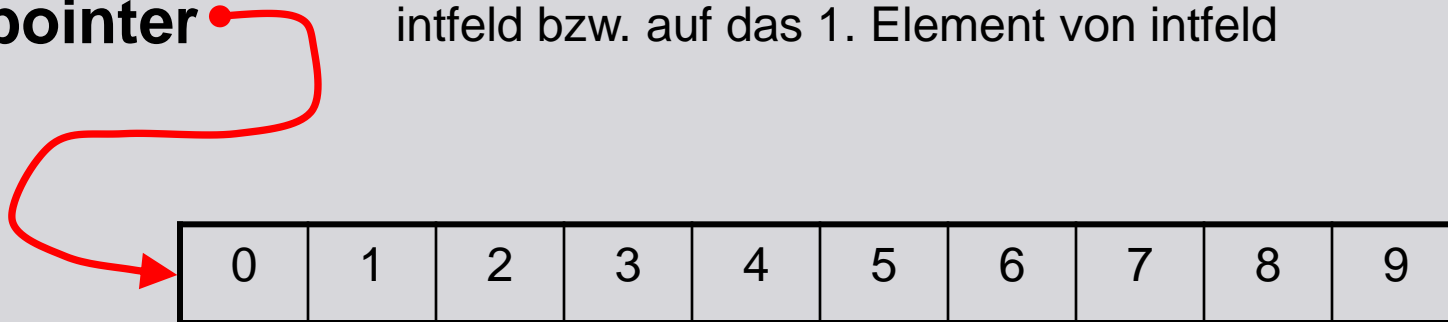
Pointer

2. Pointer und Arrays



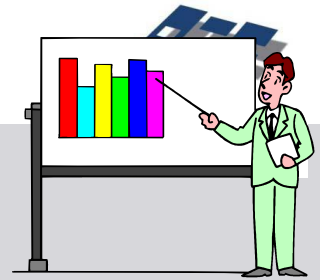
ipointer

nach `ipointer = intfeld` zeigt `ipointer` auf
`intfeld` bzw. auf das 1. Element von `intfeld`



intfeld

Pointer



2. Pointer und Arrays

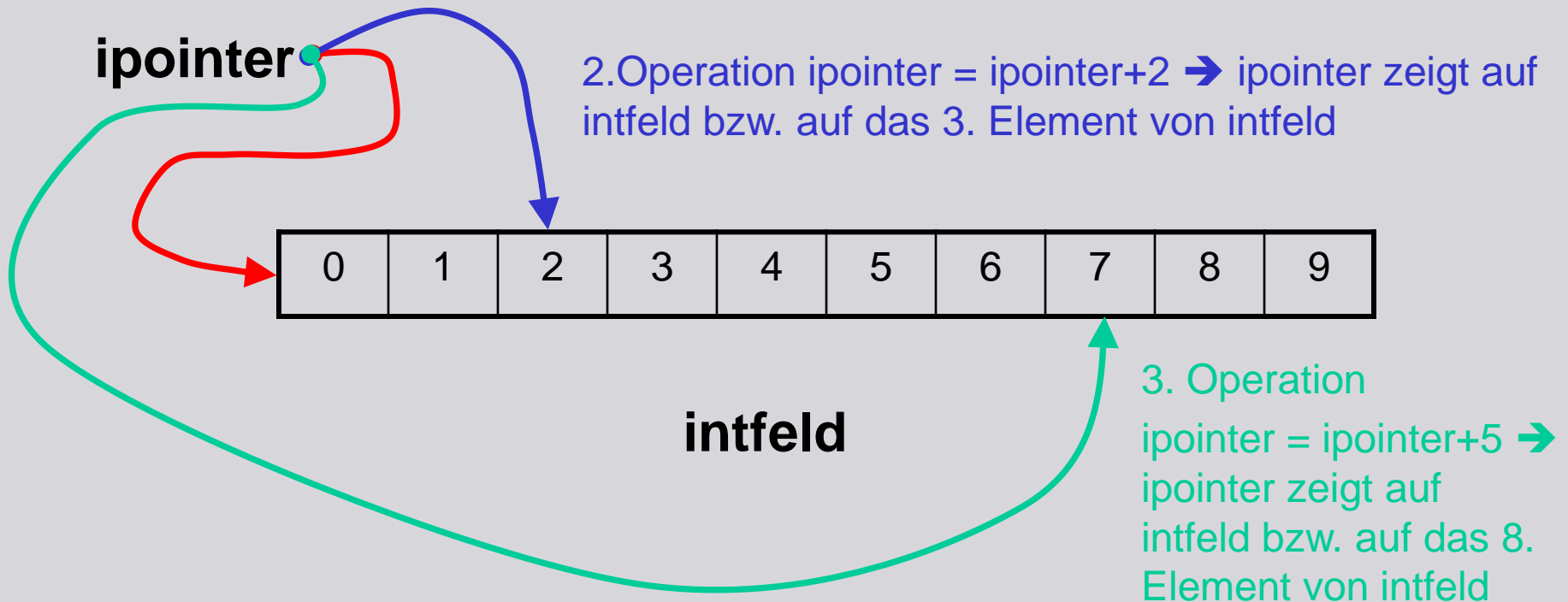
- durch Pointerarithmetik kann auf über `ipointer` auf die Elemente von `intfeld` zugegriffen werden.
- `ipointer+1` zeigt auf das 2. Element von `intfeld`
- `ipointer+2` zeigt auf das 3. Element von `intfeld`
- `ipointer + k` zeigt auf das $(k+1)$ -te Element von `intfeld`
- dies gilt für alle Arraytypen, unabhängig vom Typ des Arrays
- der Pointer wird um so viele Bytes erhöht, wie der Grundtyp des Arrays Bytes umfasst

Pointer

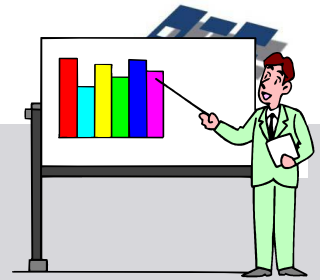


2. Pointer und Vektoren (Arrays)

1. Operation $\text{ipointer} = \text{intfeld}$ → ipointer zeigt auf intfeld bzw. auf das 1. Element von intfeld



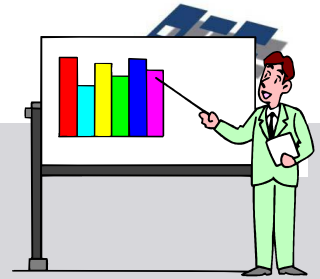
Pointer



2. Pointer und Arrays

- mittels dieser Pointerarithmetik kann ein Array durchlaufen werden, ohne die Verwendung des Arrayindex
- Pointerarithmetik kann auch ohne Arrays verwendet werden. Dies sollte allerdings mit großer Vorsicht durchgeführt werden!

Pointer



3. Pointer und Weiteres

- **Pointer können auch vom Typ Pointer auf ... sein:**

```
int **ptr_intptr; /* Pointer auf Pointer auf int */  
int i = 5;  
int *intptr;  
intptr = &i;  
ptr_intptr = &intptr;
```

Pointer



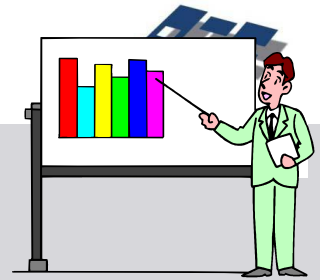
3. Pointer und Weiteres

- Pointer können auch vom Typ Pointer auf ... sein

**Pointer auf
Pointer**

Adr.	Speicher*)
1	
2	
3	
4	7153
5	
...	
7153	9999
...	
9999	Wert
10000	

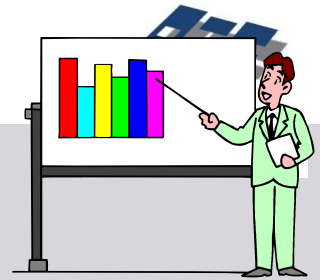
Pointer



3. Pointer und Weiteres

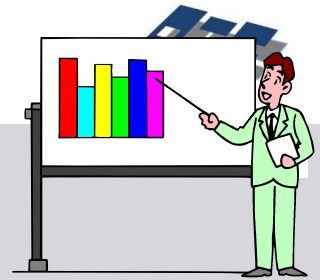
- **Pointer können, wie Variable von normalen Typen, gecastet werden:**
`int *intpointer;`
`unsigned int natzahl = 5;`
`(unsigned int*) intpointer = &natzahl;`
- **Analog zum Casten von Variablen muss Pointercasten vorsichtig durchgeführt werden!**

Pointer



- 4. Pointer und Dynamische Speicherverwaltung**
 - **Während der Laufzeit eines Programms wird der Arbeitsspeicher dynamisch benutzt**
 - **Speicherinhalte ändern sich**
 - **Menge des benutzten Speichers ändert sich (Parameter, lokale Variable)**
 - **Automatisch, implizit durch Compiler (bzw. Laufzeitsystem)**
 - ➔ **Möglichkeit der expliziten dynamischen Speicherverwaltung durch Programmierer**

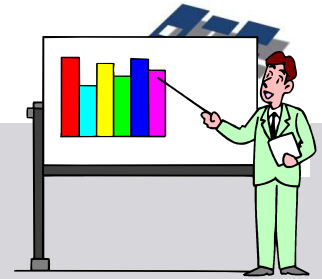
Pointer



4. Pointer und Dynamische Speicherverwaltung

- Dynamische Speicherverwaltung explizit durch den SW Entwickler*)
- Aufteilung des Arbeitsspeichers in
 - Stack (=„Stapel“, verwaltet durch den Compiler)
 - globale, lokale Variable
 - Parameter
 - Heap (=„Halde“, verwaltet durch den Entwickler)
 - Speicherplatz für dynamische Datenstrukturen

Pointer



4. Pointer und Dynamische Speicherverwaltung Arbeitsweise Stack (Wiederholung)

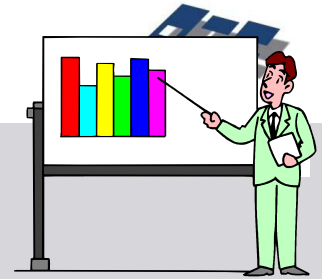
```
int j = 7;  
...  
{ int i = 5;  
  ...  
}  
j = 10;
```

Variable j
wird angelegt

Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	
9999	7
10000	...

Stack wächst von
„unten nach oben“

Pointer



4. Pointer und Dynamische Speicherverwaltung Arbeitsweise Stack (Wiederholung)

```
int j = 7;  
...  
{ int i = 5;  
  ...  
}  
j = 10;
```

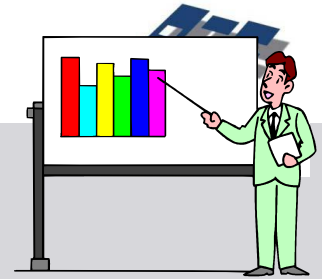
Variable i
wird angelegt

i
j

Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	5
9999	7
10000	

Stack wächst von
„unten nach oben“

Pointer



4. Pointer und Dynamische Speicherverwaltung Arbeitsweise Stack (Wiederholung)

```
int j = 7;  
...  
{ int i = 5;  
...  
}
```

Variable i
wird
gelöscht,
Stack
freigegeben


```
j = 10;
```

j

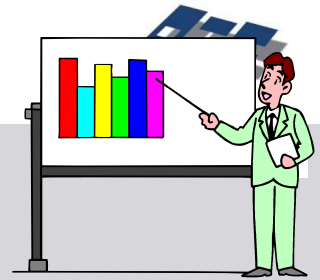


Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	
9999	7
10000	

Stack „schrumpft“

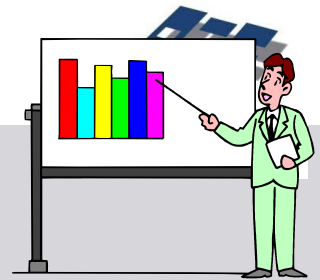


Pointer



- ## 4. Pointer und Dynamische Speicherverwaltung
- bedarfsgerechte Nutzung des vorhandenen Speichers (nur so viel Speicher verbrauchen, wie aktuell benötigt wird)
 - Anfordern von Speicherplatz bei Bedarf
 - Freigeben von nicht mehr benötigten Speicher
 - Verwendung für dynamische Datenstrukturen, d.h. Datenstrukturen mit variabler Größe
- Effiziente Nutzung des Speichers (😊)
- Verwaltung liegt beim Entwickler (😐)

Pointer



4. Pointer und Dynamische Speicherverwaltung

Heap wächst von
„oben nach unten“

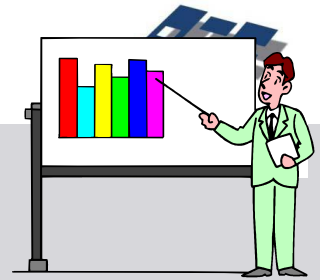
Grenze des
belegten Stack
(„pulsiert“)

Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	
9999	
10000	

Grenze des
belegten Heap
(„pulsiert“)

Stack wächst von
„unten nach oben“

Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Speicherplatzanforderung mit malloc()**
 - **Bibliotheksfunktion `void* malloc(Anzahl Bytes)`**
 - **liefert einen Pointer auf einen Speicherbereich der benötigten Größe (oder Fehlercode, falls kein Speicher der geforderten Größe mehr verfügbar)**
 - **Gutfall: Speicherplatz wird reserviert und kann beschrieben werden**
 - **malloc() liefert einen typlosen (`void*`) Pointer zurück, dieser muß auf den richtigen Typ gecastet werden**

Pointer



4. Pointer und Dynamische Speicherverwaltung

- Speicherplatzanforderung mit `malloc()`
- Anzahl Bytes konkret angeben

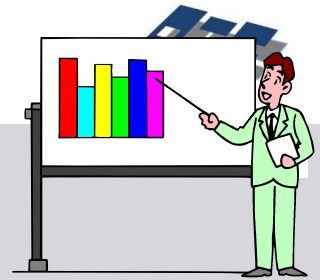
`intptr = (int*) malloc(Byteanzahl);`

↑
Cast auf
richtigen
Pointertyp

↑
reserviert
Speicherplatz

↑
gibt an, wie viele
Bytes reserviert
werden sollen

Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Speicherplatzanforderung mit malloc()**
 - **Anzahl Bytes mittels sizeof() Bibliotheksfunktion ermitteln lassen (meist bessere Lösung!)**
 - **sizeof wird mit dem Typ aufgerufen, auf den der Pointer zeigen soll, z.B.**

intptr = (int*) malloc(sizeof(int));

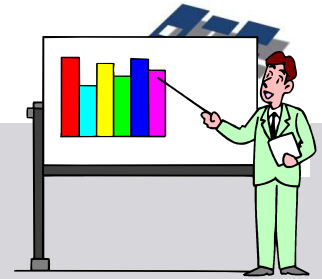
Cast auf
richtigen
Pointertyp

reserviert
Speicherplatz

gibt an, wie viele
Bytes reserviert
werden sollen

*)sizeof(...) kann auch mit einer Variablen aufgerufen werden und liefert den Speicherplatzverbrauch dieser Variablen

Pointer



4. Pointer und Dynamische Speicherverwaltung

`malloc(sizeof(int));`
liefert Adresse
3555 zurück

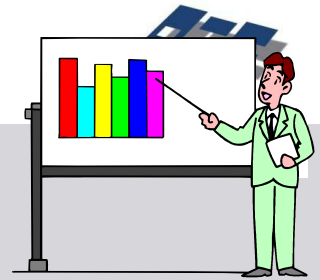
Grenze des
belegten Stack

Adr.	Arb.-Speicher
1	
2	
3	
...	
3555	
...	
...	
9998	
9999	
10000	

Grenze des
belegten Heap
vor `malloc(...)`

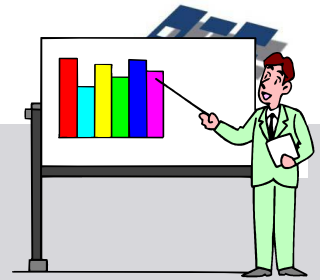
Grenze des
belegten Heap
nach `malloc(...)`

Pointer



- 4. **Pointer und Dynamische Speicherverwaltung**
 - **Speicherplatzanforderung mit malloc()**
 - **Ergebnis von malloc(...) immer abfragen!**
 - **Möglichst den Compiler die Größe des benötigten Speichers ermitteln lassen: malloc(sizeof(complex)) anstatt (malloc(8))**
 - **Bei kleinen Speichern (Mikrocontroller) dynamische Speicherverwaltung nur sehr vorsichtig (oder gar nicht) einsetzen!**

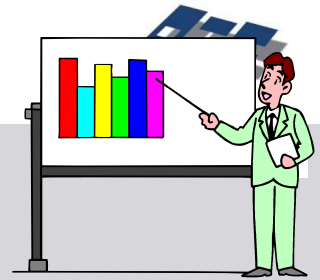
Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Speicherplatzanforderung mit new in C++**
 - **reserviert (wie malloc) Speicher der benötigten Größe**
 - **kann auch für C Typen verwendet werden**
`int* intptr = new int;`
`complex* cpointer = new complex;`
 - **Cast auf den richtigen Pointertyp ist nicht erforderlich**
 - **ruft für Klassen implizit einen Konstruktor auf**

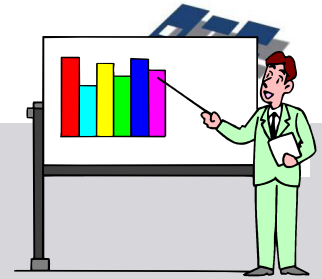
Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Speicherplatzfreigabe mit free()**
 - Bibliotheksfunktion `void free(pointer)`
 - gibt den Speicherplatz, der zuvor mit `malloc(...)` reserviert wurde, wieder frei.
 - Freigabe heißt, dass der Speicher für erneute Speicherplatzreservierungen wieder zur Verfügung steht.
 - Der freizugebende Speicher muss nicht am Ende des Heaps liegen → entstehende Lücken werden vom Compiler soweit möglich wieder genutzt

Pointer



4. Pointer und Dynamische Speicherverwaltung

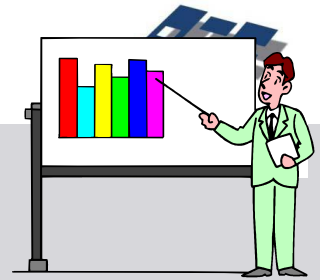
intptr
free(intptr) gibt
belegten
Speicherplatz wieder
frei

Grenze des
belegten Stack

Adr.	Arb.-Speicher
1	
2	
...	
2715	
...	
3555	
...	
9998	
9999	
10000	

Grenze des
belegten Heap

Pointer



4. Pointer und Dynamische Speicherverwaltung

- Speicherplatzfreigabe mit `free()`
 - Ein Pointer, dessen Speicherplatz freigegeben wurde, darf nicht mehr dereferenziert oder zugewiesen werden:



...

```
free (intptr);
```

```
*intptr = 5; /* verboten */
```

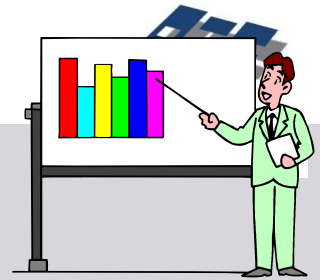
```
intptr2 = intptr; /* verboten */
```

...

```
intptr = intptr3; /* OK */
```

```
intptr = (int*) malloc(...); /* OK */
```

Pointer



4. Pointer und Dynamische Speicherverwaltung

- Speicherplatzfreigabe mit `free()`
 - Die Speicherplatzfreigabe soll mit dem gleichen Pointer erfolgen, mit dem der Speicher angefordert wurde:

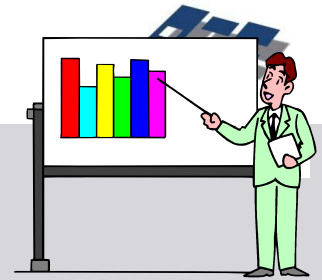
```
intptr = (int*) malloc(sizeof(int));
```

```
...
```

```
intptr = intptr2;
```

```
free (intptr); /* vermeiden */
```

Pointer



4. Pointer und Dynamische Speicherverwaltung

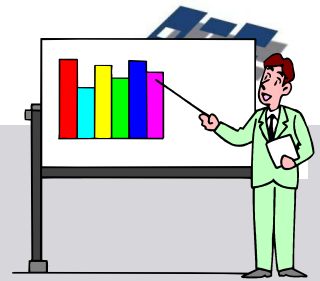
- Speicherplatzfreigabe mit `free()`
 - `free` darf nicht am gleichen Pointer mehrfach hintereinander aufgerufen werden (ohne zwischenzeitliches Anfordern von Speicher):



```
...  
free (intptr);  
free (intptr); /* verboten */
```

```
...  
free (intptr); /* OK */  
intptr = malloc(...);  
free (intptr); /* OK */
```

Pointer



4. Pointer und Dynamische Speicherverwaltung

- Speicherplatzfreigabe mit `free()`
 - Speicherplatzanforderungen mit `new` und Freigaben mit `delete` dürfen nicht gemischt werden:

...

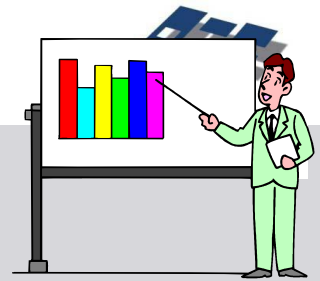
```
int* intptr = new int;
```

...

```
free (intptr); /* verboten */
```



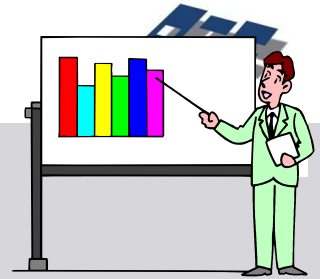
Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Speicherplatzfreigabe mit delete in C++**
 - gibt (wie free(...)) Speicher frei
 - kann für C Typen verwendet werden
`delete intptr;`
`delete cpointer;`
 - ruft für Klassen implizit einen Destruktor auf

Pointer



4. Pointer und Dynamische Speicherverwaltung

- Speicherplatzfreigabe mit delete in C++
 - Speicherplatzanforderungen mit malloc() und Freigaben mit delete dürfen nicht gemischt werden:



```
int* intptr = malloc(...);  
delete intptr; /* verboten */
```

Pointer



- 4. **Pointer und Dynamische Speicherverwaltung**
 - **Memory Leaks („Speicherlecks“)**
 - **Speicherbereich im Heap, der reserviert, aber nicht mehr zugänglich (über Pointer erreichbar) ist**
 - **Problem bei unsauberer Speicherverwaltung**
 - ➔ **führt zu Speichermangel**
 - ➔ **abnormale Programmbeendigung**

Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Memory Leaks („Speicherlecks“)**
 - **Entstehung:**

```
intptr = (int*) malloc(sizeof(int));  
/* intptr zeigt auf reservierten Speicherbereich */  
/* im Heap. intptr ist der einzige Zugang zu      */  
/* diesem Bereich                                  */  
intptr = NULL; /* kann auch anderen Wert sein */  
/* Speicherbereich kann nicht mehr mittels      */  
/* free freigegeben werden */
```

Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Memory Leaks („Speicherlecks“)**

intptr = (int*)
malloc(sizeof(int));
liefert Adresse 3555
zurück.

intptr = NULL;
zerstört Zugang zu
reserviertem Speicher

Adr.	Arb.-Speicher
1	
2	
3	
...	
3555	
...	
...	
9998	
9999	
10000	

Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Memory Leaks („Speicherlecks“)**

Wiederholtes Erzeugen von Memory Leaks führt zur „Vermüllung“ und letztlich zum „Verlust“ des Heaps

Adr.	Arb.-Speicher
1	
2	
3	
...	
3555	
...	
...	
9998	
9999	
10000	

Pointer



4. Pointer und Dynamische Speicherverwaltung

- **Memory Leaks („Speicherlecks“)**
- **Vermeidung:**
 - **Dynamische Speicherverwaltung nur wenn wirklich notwendig**
 - **sorgfältige Programmierung der Dynamischen Speicherverwaltung (d.h. Speicherplatzanforderung und Freigabe nur an wenigen definierten Stellen**
 - **Einsatz von Programmiersprachen mit eingebauten „Müllsammlern“ (Garbage Collectoren), z.B. Java**

Pointer

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Funktionen

5. Funktionen und Prozeduren

- in der Mathematik:
 - seit langem bekannt

$$f(x) := x^2$$

$$y = f(x)$$

Abhängig von einem Parameter wird ein Funktionswert (immer auf die gleiche Art) berechnet.

Funktionen

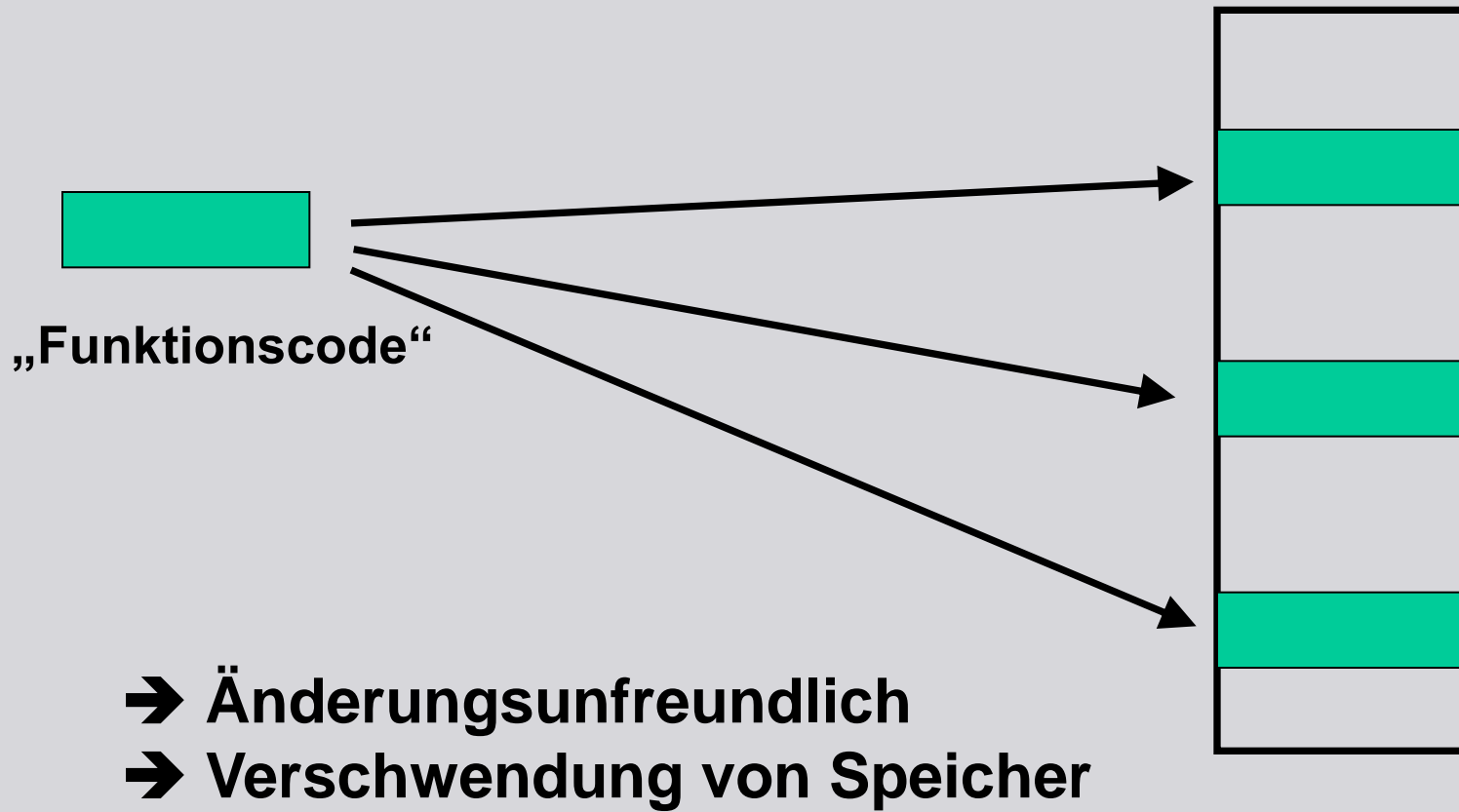
5. Funktionen und Prozeduren

- In der Informatik:
 - gegeben: „Stück Software“, das ein Ergebnis liefert und immer wieder mit verschiedenen Werten an verschiedenen Stellen ausgeführt werden soll
 - Lösung 1: überall dort wo benötigt einkopieren
 - ➔ Änderungsunfreundlich
 - ➔ Verschwendung von Speicher

Funktionen

5. Funktionen und Prozeduren

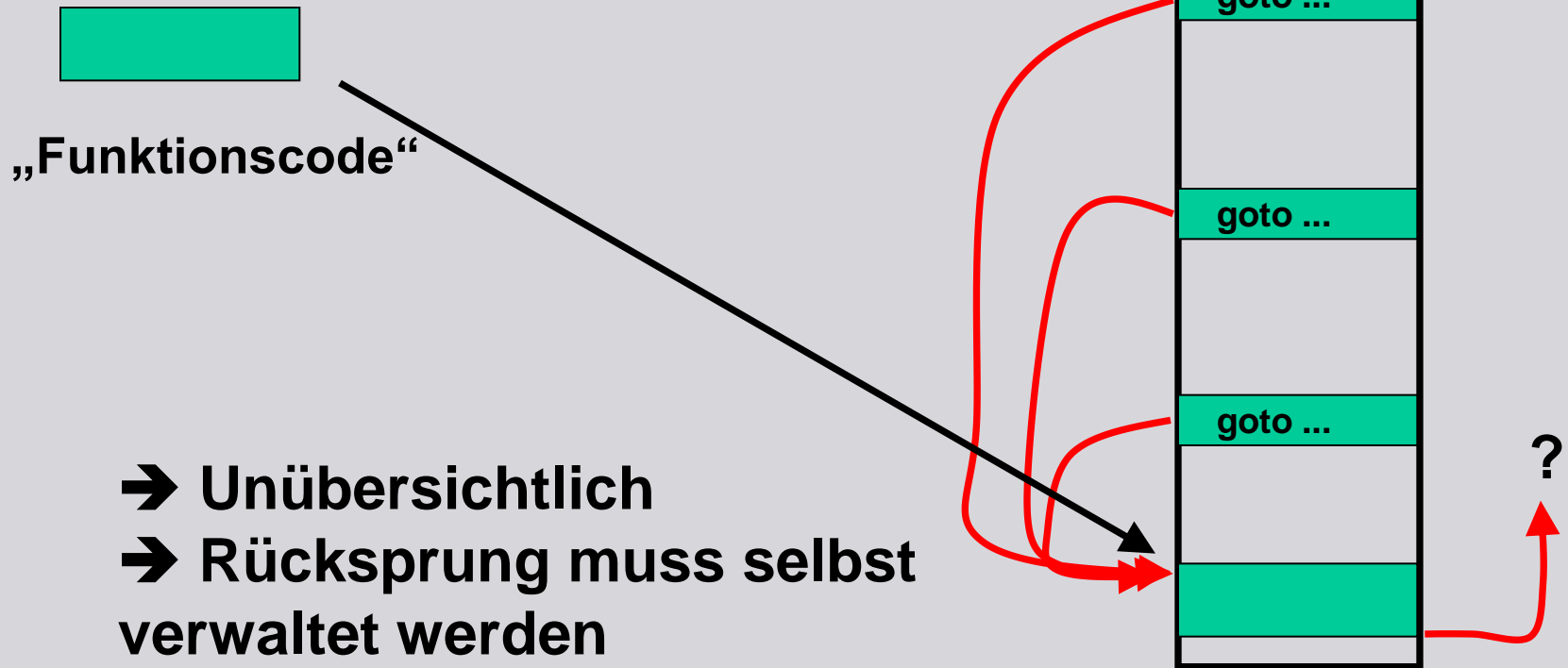
- Lösung 1: überall dort wo benötigt einkopieren



Funktionen

5. Funktionen und Prozeduren

- Lösung 2: mit goto's



Funktionen

5. Funktionen und Prozeduren

- **Lösung 3: Definieren als Funktion**
 - **„Stück Software“ bekommt einen eindeutigen Namen, unter dem es aufgerufen (d.h. ausgeführt wird), einen Ergebnistyp und ggf. eine Reihe von Parametern, die beim Aufruf übergeben werden.**
 - **aus $f(x) = x^2$ wird `float quadrat(float x)`**

Funktionen

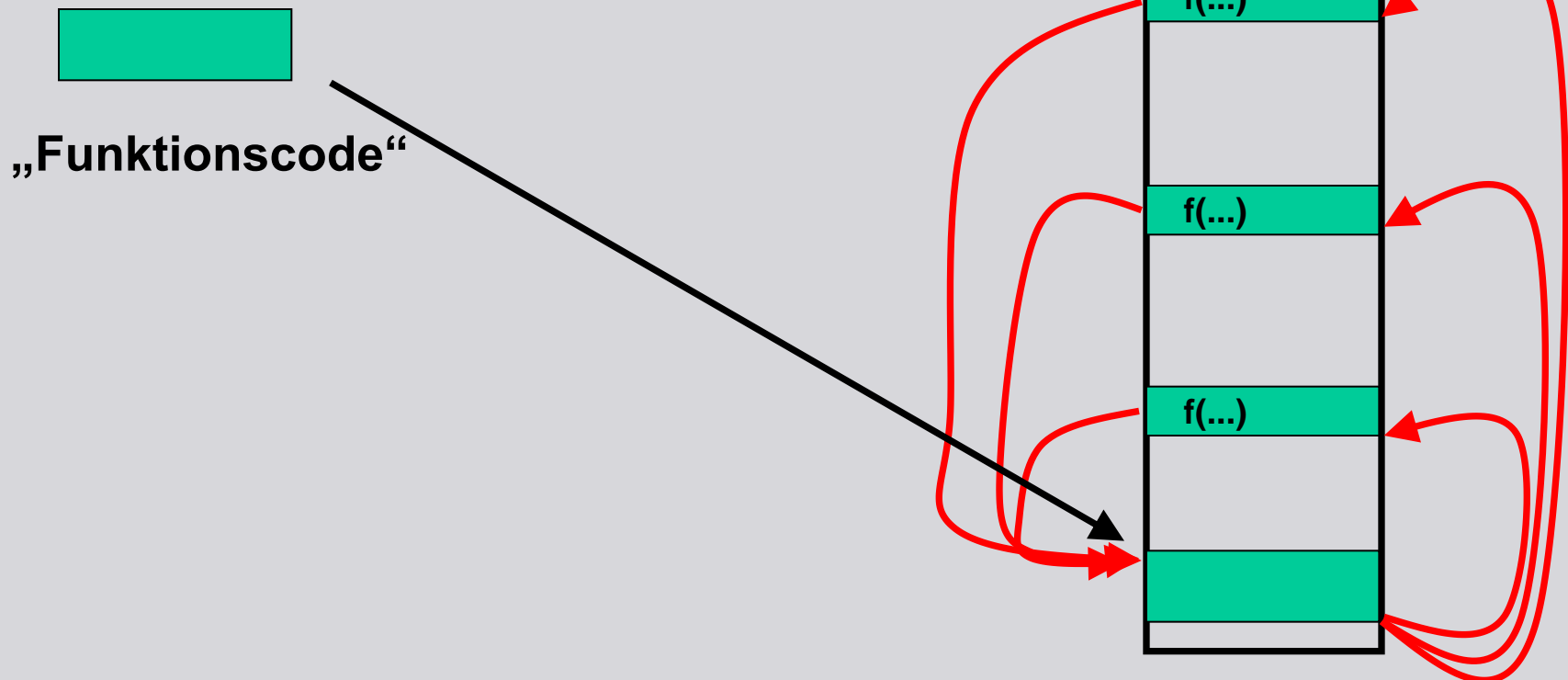
5. Funktionen und Prozeduren

- **Lösung 3: Definieren als Funktion**
 - **Funktion kann beliebig oft aufgerufen werden**
 - **Der Aufrufer ist nur noch verantwortlich für die richtigen Parameter und für das „Abholen“ des Ergebnisses**
 - **Der Sprung in die Funktion und der Rücksprung geschehen automatisch**

Funktionen

5. Funktionen und Prozeduren

- Lösung 3: Definieren als Funktion



Funktionen

5. Funktionen und Prozeduren

- **Syntaktischer Aufbau einer Funktion in C:**
Ergebnistyp Funktionsname (Parameterliste)
{ /* Funktionsrumpf */ }
- **Ergebnistyp legt fest, was die Funktion als Ergebnis zurückgibt**
- **Funktionsname muss im Programm eindeutig sein (inkl. Ergebnistyp und Parameterliste)**
- **Ergebnistyp und Parameterliste werden auch als Signatur der Funktion bezeichnet**
- **Der Aufrufer der Funktion und der Compiler benötigen die Signatur**

Funktionen

5. Funktionen und Prozeduren

- **Syntaktischer Aufbau einer Funktion:**
Ergebnistyp Funktionsname (Parameterliste)
{ /* Funktionsrumpf */ }
- **Die Signatur legt das Abbildungsverhalten der Funktion fest: $f: P_1 \times P_2 \times \dots \times P_n \rightarrow E$**
mit $P_i :=$ Wertmenge des Parameters i , E
Wertmenge des Ergebnisses
- **Es kann auch Funktionen geben, die keine Parameter haben. Diese liefern (theoretisch) immer das selbe Ergebnis**

Funktionen

5. Funktionen und Prozeduren

- **Syntaktischer Aufbau einer Funktion in C:**
Ergebnistyp Funktionsname (Parameterliste)
{ /* Funktionsrumpf */ }
- **Der Funktionsrumpf enthält u.a.:**
 - **lokale Variable, Konstante, Parameter**
Diese sind nur innerhalb der Funktion existent und bekannt!
Namensgleiche globale Namen werden verschattet!
 - **Anweisungen inkl. Aufrufe von Funktionen, die u.a. die Parameter verarbeiten und abhängig von Parameterwerten ein Ergebnis berechnen**

Funktionen

5. Funktionen und Prozeduren

- **Syntaktischer Aufbau einer Funktion:**
Ergebnistyp Funktionsname (Parameterliste)
{ /* Funktionsrumpf */ }
- **Der Funktionsrumpf enthält u.a.:**
 - **eine (oder mehrere) return Anweisungen zur Rückgabe des Funktionsergebnisses, Beendigung der Funktion und Rücksprung an die Aufrufstelle**
- **Der Funktionsrumpf enthält in C/C++ keine weiteren Funktionsdefinitionen. Andere Programmiersprachen erlauben das Schachteln von Funktionen**

Funktionen

5. Funktionen und Prozeduren

- Syntaktischer Aufbau einer Funktion, Beispiele:

```
float quadrat (float x)
{ return x * x; };
```

```
int max (int a, int b)
{ if (a < b)
  return b;
else return a;
};
```

Ergebnis-
rückgabe,
Rücksprung

formale Parameter

Funktionen

5. Funktionen und Prozeduren

- Aufruf einer Funktion, Beispiele:

```
float p = 1.2;  
float q = quadrat (p);
```

aktuelle Parameter

```
int i1 = 10;  
int i2 = 20;  
int max_von_i1_und_i2 = max (i1, i2)
```

Ergebnis-
übergabe

Funktionen

5. Funktionen und Prozeduren

- **lokale Variable, Gültigkeitsbereich, Verschattung, Beispiele:**

```
...  
float rechenergebnis = 0;  
...  
float a_hoch_n (float a, int n)  
{ int zaehler;  
  float rechenergebnis = 1;  
  for (zaehler = 1; zaehler <=n; zaehler++)  
    rechenergebnis = rechenergebnis * a;  
  return rechenergebnis;  
};
```

globale Variable wird durch lokale Variable gleichen Namens verschattet.

Die globale Variable ist dadurch innerhalb der Funktion nicht bekannt

Funktionen

5. Funktionen und Prozeduren

- **Regeln / Empfehlungen zur Verwendung von Funktionen:**
 - **Aussagekräftige Funktionsnamen**
 - **Aussagekräftige Parameternamen**
 - **„kleine“ Funktionen (max. 100 Codezeilen)**
 - **alle formalen Parameter mit Werten besetzen**
 - **Parametertypen beachten**
 - **sinnvolles Funktionsergebnis zurückgeben**
 - **Funktionsergebnis abholen (z.B. Rückgabe Fehlercode)**
 - **wenige return Anweisungen**

Funktionen

5. Funktionen und Prozeduren

- **Regeln / Empfehlungen zur Verwendung von Funktionen:**
 - **möglichst parametrieren, keine globalen Variablen innerhalb von Funktionen lesend oder schreibend verwenden**

Funktionen

5. Funktionen und Prozeduren

- **Regeln / Empfehlungen, Gegenbeispiele:**

```
int max (int a, b)
{ if (a < b)
  return b;
  else return a;
};
```

```
float f1 = 3.5; float f2 = 7.7;
float maxf = (max(f1,f2));
```

implizite Typkonvertierung
auch bei Parametern und
Ergebnis

```
int i1 = 5; int i2 = 10;
max(i1, i2);
```

Ergebnis wird nicht abgeholt

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter:
 - Beim Funktionsaufruf werden die formalen Parameter der Funktion durch die aktuellen Parameter „ersetzt“
 - Konstante, z.B. `int m = min(5,10);`
„Call by Value“
 - Variable eines Standardtyps (`int`, `float`, ...) oder eines selbst definierten Typs,
z.B. `int m1 = 5; int m2 = 10; int m = min(i1, i2);`
`double betrag(complex c);`
„Call by Name“

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter:

- Pointer, z.B.

```
int min_ueber_Pointer (int *ip1, int *ip2)
/* gibt die kleinere Zahl zurück auf die die */
/* Pointer zeigen */
{ if (*ip1 < *ip2)
    return *ip1;
  else return *ip2;
};
```

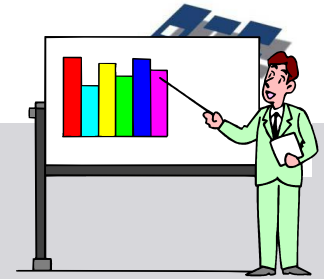
„Call by Reference“

Funktionen

5. Funktionen und Prozeduren

- **Funktionsparameter:**
 - **werden grundsätzlich auf dem Stack übergeben**
 - **analog Variable**
 - **Aufruf der Funktion: aktuelle Parameter werden auf den Stack kopiert**
 - **Beenden einer Funktion: Stackbereich der Parameter wird wieder freigegeben**

Pointer



5. Funktionen und Prozeduren Parameterübergabe auf dem Stack

```
int f(int i)
{ ...
};
...
{
  ... = f(5);
}:
```

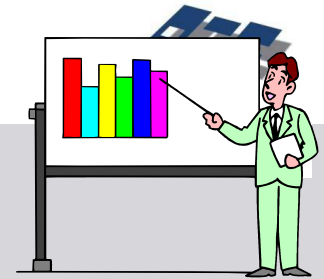
Parameter i
wird beim
Aufruf der
Funktion f auf
dem Stack
angelegt.
aktueller Wert
(hier 5) wird
gespeichert

Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	5
9999	,,,
10000	,,,



Stack wächst von
„unten nach oben“

Pointer



5. Funktionen und Prozeduren Parameterübergabe auf dem Stack

```
int f(int i)
{ ...
};
...
{
  ... = f(5);
  }:
```

Nach
Beendigung der
Funktion f wird
der Stack-
bereich des
Parameters i
wieder frei-
gegeben

Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	
9999	'''
10000	'''



Stack „schrumpft“

Funktionen

5. Funktionen und Prozeduren

- **Funktionsparameter:**
 - **können innerhalb einer Funktion entsprechend ihrer Deklaration auch als lokale Variable schreibend verwendet werden.**
- ➔ Diese Möglichkeit sollte aber nicht genutzt werden, da Parameter zunächst einen reinen „Input-Charakter“ haben (Ausnahme: Parameter, die per Pointer übergeben werden).**

Funktionen

5. Funktionen und Prozeduren

- **Funktionsparameter:**
 - **können innerhalb einer Funktion entsprechend ihrer Deklaration auch als lokale Variable schreibend verwendet werden.**

➔ Das Überschreiben der Parameter hat keine „Außenwirkung, d.h. die Variablen, die als aktuelle Parameter verwendet werden, werden nicht verändert!

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter: Gegenbeispiele:

```
double betrag (complex c)
{ c.real = c.real * c.real;
  c.imag = c.imag * c.imag;
  return sqrt(c.real+c.imag);
};
```

Missbräuchliche Verwendung
eines Parameters

```
...
main(...)
{ complex c_global;
  double b;
  c_global.real = 3;
  c_global.imag = 4;
  b = betrag(c);
  ...
};
```

Nach Ausführung der Funktion
betrag hat c_global immer noch
die Werte 3 (Realteil) und 4
(Imaginärteil)

Funktionen

5. Funktionen und Prozeduren

- **Arrays als Funktionsparameter:**
 - **Übergabe immer per Referenz, Beispiel:**

```
int addierearray(int a[5])  
{ int i;  
  int erg = 0;  
  for (i=0; i<5;i++)  
    erg = erg + a[i];  
  return erg;  
};
```

Funktionen

5. Funktionen und Prozeduren

- **Arrays als Funktionsparameter:**
 - **Übergabe immer per Referenz, Beispiel:**

```
void initialisierearray(int a[5])
```

```
{ int i;
```

```
  for (i=0; i<5;i++)
```

```
    a[i] = 0;
```

```
  return;
```

```
};
```

```
...
```

```
int f[5] = {1,2,3,4,5};
```

```
initialisierearray(f);
```

```
/* alle Elemente von f haben hier den Wert 0 */
```

Funktionen

5. Funktionen und Prozeduren

- **Funktionsergebnis, Rückgabe:**
 - **als Konstante:** `return 5;`
 - **als Variable eines Standardtyps (int, float, ...) oder eines selbst definierten Typs, z.B.**
`int m; ... /* Berechnung eines Werts für m */`
`return m;`
`complex c; ... /* Berechnung eines Werts für m */`
`return c;`
 - **als Ausdruck, z.B.**
`return m+5; return f(x);`
- ➔ **Es wird immer ein Wert (einer Konstanten, einer Variablen, eines Ausdrucks) zurückgegeben, nicht die Variable oder die Konstante!**

Funktionen

5. Funktionen und Prozeduren

- **Funktionsergebnis, Rückgabe:**

- **als Referenz, z.B:**

```
struktur* f(...)  
{ struktur *s; s = (struktur*) malloc(sizeof(struktur));  
  /* Berechnung eines Werts für *s */  
  return s;  
};
```

Bei Rückgabe komplexer bzw. dynamischer Datenstrukturen. Auch hier Rückgabe eines Werts (Pointer).

Sicherstellen, dass der angeforderte Speicher korrekt verwaltet wird!

Funktionen

5. Funktionen und Prozeduren

- Funktionsergebnis, Rückgabe:

- als Referenz, **aber so nicht:**

```
struktur* f(...)
```

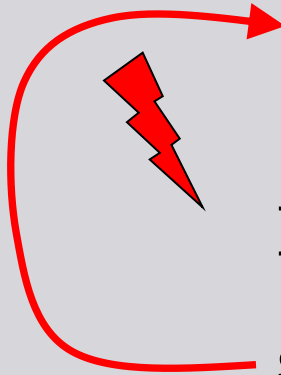
```
{ ...
```

```
  struktur s;
```

```
  /* Berechnung eines Werts für *s */
```

```
  return &s;
```

```
};
```



s ist außerhalb von f nicht bekannt, somit ist der Inhalt von s auch nicht mehr gültig!

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter:
 - Implementierung von Funktionen, die mehr als einen Ergebniswert zurückliefern?

Beispiel:

Funktion, die den Arbeitszustand eines Verbrennungsmotors liefern soll.

Der Zustand beinhaltet:

- aktuelle Drehzahl
- Öldruck
- Öltemperatur
- Wassertemperatur

Funktionen

5. Funktionen und Prozeduren

- **Funktionsparameter:**
 - **Funktion, die den Arbeitszustand eines Verbrennungsmotors liefern soll.**
Lösung 1: Definieren eines Typs Motorzustand mit den entsprechenden Komponenten, Rückgabe per return

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter:
 - **Lösung 1: Definieren eines Typs Motorzustand, Rückgabe per return**

```
typedef struct m_zustand  
{ int Drehzahl; int Öldruck;  
  int Wassertemp; int Öltemp;  
};
```

```
m_zustand motorzustand ()  
{ m_zustand akt_zustand;  
  akt_zustand.Drehzahl = ...; akt_zustand.Öldruck = ...  
  ...  
  return akt_zustand;  
}
```

Funktionen

5. Funktionen und Prozeduren

- **Funktionsparameter:**
 - **Funktion, die den Arbeitszustand eines Verbrennungsmotors liefern soll.**
Lösung 2: Übergabe des Motorzustands in 4 Variablen, die per Referenz übergeben werden.
(Output Variable)
→ erspart zusätzlichen Typ und Zugriffe über Komponenten

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter:
 - Lösung 2: Übergabe des Motorzustands in 4 Output Variablen, Definition

Keine „formalen“
Funktionsergebnis
ausgedrückt durch
void

```
void motorzustand (int* Dz, int* Öd, int* Wt, int* Öt)
{ ...
  *Dz = ... ;
  *Öd = ...;
  *Wt = ... ;
  *Öt = ...;
  ...
  return;
}
```

Definition von formalen
Referenzparametern

Funktionen

5. Funktionen und Prozeduren

- Funktionsparameter:
 - **Lösung 2: Übergabe des Motorzustands in 4 Output Variablen, Aufruf**

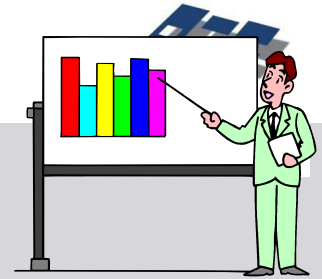
int Drehzahl;
int Öldruck;
int Wassertemp;
int Öltemp;
...
motorzustand (&Drehzahl, &Öldruck, &Wassertemp, &Öltemp);
...

Adressen von Variablen
als aktuelle Parameter



Funktionen

5. Funktionen und Prozeduren



motorzustand (
&Drehzahl,
&Öldruck,
&Wassertemp,
&Öltemp);

Funktion schreibt
direkt über die
Adressen der
Variablen in den
Speicher

Adr.	Arb.-Speicher
1	
2	
3	
...	
3555	Drehzahl
3556	Öldruck
3557	Wassertemp
3558	Öltemp
...	
10000	

Funktionen

5. Funktionen und Prozeduren

- **Prozeduren:**
 - **gegeben: „Stück Software“, das immer wieder mit verschiedenen Werten an verschiedenen Stellen ausgeführt werden soll**
 - **gegeben: „Stück Software“, das mehrere Ergebnis zurückliefert und das immer wieder mit verschiedenen Werten an verschiedenen Stellen ausgeführt werden soll**
- ➔ **„Funktionen“ dieser Art werden auch als Prozeduren bezeichnet. Andersherum heißen Funktionen manchmal auch Funktionsprozeduren**

Funktionen

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Funktionspointer

6. Funktionenspointer

- **Pointer können in C nicht nur auf Daten, sondern auch auf Funktionen zeigen.**
- **Syntax analog zu Pointer auf einen Datentyp:**

int (*fpointer) (int);

zeigt auf eine Funktion mit Returntyp int und einem int Parameter *)

***) nicht `int* f (int) ! `()`` bindet stärker als ``*``**

Funktionspointer

6. Funktionenspointer

- **Wertzuweisung und Dereferenzierung ähnlich wie bei „herkömmlichen“ Pointern:**

```
int a (int z)
{ if (z <0)
    return -z;
  else return z;
};
```

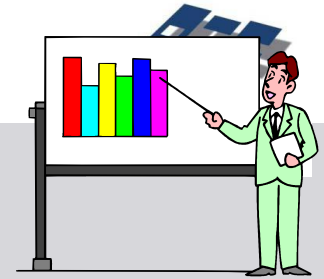
```
int (*fpointer) (int); /* Funktionspointer */
```

```
fpointer = &a; /* f wird die Adresse von a zugewiesen */
```

```
int b = (*fpointer)(-10); /* Die Funktion, auf die f zeigt wird
aufgerufen */
```

Funktionspointer

6. Funktionenspointer

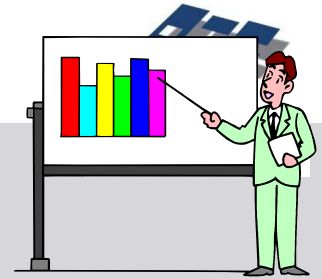


Anfangsadresse
von a
beim Aufruf von a
springt der Ablauf
in den Code von a
ab dieser Adresse

Adr.	Programmspeicher
1	
2	
3	
...	
3555	Int a (int z)
...	{ ...
...	...
3700	... }
...	
10000	

Funktionspointer

6. Funktionenspointer



Anfangsadresse
von a

Durch `fpointer = a;`
zeigt `fpointer` auch
auf die Anfangs-
adresse von a

beim Aufruf von
`fpointer(10)` springt
der Ablauf in den
Code, auf den
`fpointer` zeigt

Adr.	Programmspeicher
1	
2	
3	
...	
3555	Int a (int z)
...	{ ...
...	...
3700	... }
...	
10000	

Funktionspointer

Zum Schluss dieses Abschnitts ...

Noch Fragen ??

Dateien

7. Dateien

Datentyp FILE ist in C vordefiniert (in stdio.h), z.B.

```
typedef struct _iobuf
{
    char* _ptr;
    int _cnt;
    char* _base;
    int _flag;
    int _file;
    int _charbuf;
    int _bufsiz;
    char* _tmpfname;
} FILE;
```

Dateien

7. Dateien - Beispiel

Definition eines Dateizeigers:

```
FILE *datei;
```

Öffnen einer Datei in einem bestimmten Modus:

```
datei = fopen("c:\\test.txt","r") /* Öffnen z. Lesen */
```

Lesen aus einer geöffneten Datei

```
char zeile[100]; /* Lesebuffer */  
fscanf(datei,"%s", zeile);
```

Dateien

7. Dateien

Öffnen einer Datei mittels `fopen()`

`FILE* fopen(const char *pfadname, const char *modus);`

`pfadname` steht für den Dateinamen ggf. inkl. des Laufwerkpfads (`\` als `\\`angeben!), z.B. `"C:\\Ordner\\Datei1"`

`*modus` steht für den Zugriffsmodus:

`"r"` = Lesezugriff (READ)

`"rb"` = Lesen binär

`"w"` = Schreibzugriff (WRITE)

`"wb"` = Schreiben binär

`"a"` = Anfügen am Ende (Append)

`"ab"` = Anfügen binär

Dateien

7. Dateien

Öffnen einer Datei mittels fopen()

fopen liefert Pointer auf FILE-Struktur zurück falls Öffnen erfolgreich, sonst NULL-Pointer

➔ Überprüfen des Ergebnisses

```
datei = fopen( ... );  
if (datei == NULL)  
{ printf("Fehler beim Öffnen der Datei ... „);  
  ... /* weitere Fehlerbehandlung */
```


Dateien

7. Dateien

Buchstabenweises Einlesen aus Datei

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

```
char c;  
while ((c=getc(datei)) != EOF)  
{ ... };
```

Formatiertes Einlesen aus Datei

```
int fscanf(FILE *f, const char *format, ... ) /* analog zu scanf(...) */
```

Analog zu scanf() formatiertes Einlesen von Tastatur, nur dass immer geprüft werden muss, ob das Dateiende erreicht worden ist

Dateien

7. Dateien

Buchstabenweises Schreiben in eine Datei

```
int fputc(int c, FILE *datei);
```

schreibt ein Zeichen in datei, gibt Zeichen zurück oder EOF bei Fehler

Formatiertes Schreiben in eine Datei

```
int fprintf(FILE *f, const char *format, ... ) /* analog zu printf(...) */
```

Analog zu printf() formatiertes Schreiben in eine Datei

Dateien

7. Dateien Binärer Zugriff auf Dateien

➔ **Direktes Kopieren eines Bereichs des Arbeitsspeichers in eine Datei bzw. direktes Füllen eines Speicherbereichs aus einer Datei**

➔ **Öffnen der Datei im Binären Modus**

➔ **Lesen mit fread**

➔ **Schreiben mit fwrite**

Dateien

7. Dateien

Binärer Zugriff auf Dateien zum Lesen

**fread(Puffer_Adresse, Puffer_Größe, Anzahl_Puffer,
FILE *datei)**

**Puffer_Adresse = Pointer auf Datenbereich, der beim
Lesen aus Datei gefüllt wird**

**Puffer_Größe = Anzahl Bytes des Datenbereichs
(sizeof(...))**

**Anzahl_Puffer = Anzahl der Rekords, die gelesen
werden sollen**

datei = Dateipointer

Dateien

7. Dateien

Binärer Zugriff auf Dateien zum Schreiben

**fwrite(Puffer_Adresse, Puffer_Größe, Anzahl_Puffer,
FILE *datei)**

Parameter wie zuvor

Dateien

7. Dateien Schließen einer Datei

`fclose(FILE *datei)`

**Schließen nicht vergessen, ansonsten Risiko, dass
Datei nicht weiter verarbeitet werden kann.**