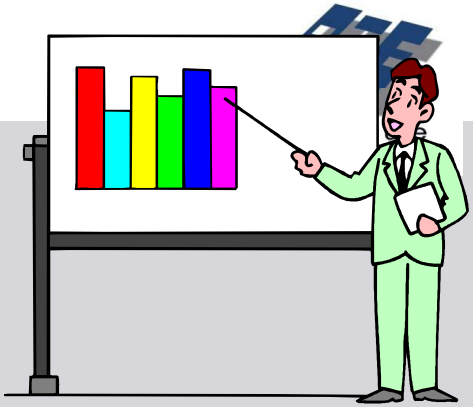
Einführung in die Programmierung

Wiederholung: Pointer

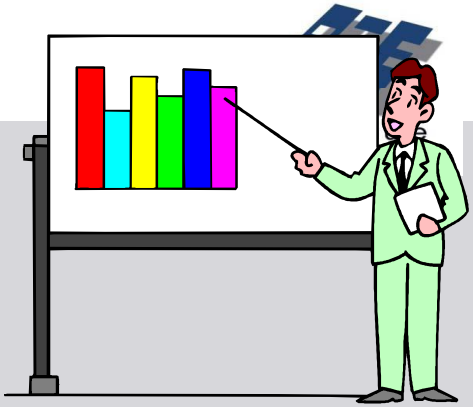
Prof. Dr. Peter Jüttner

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

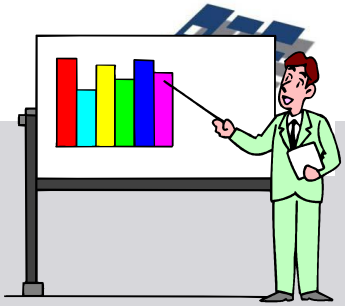
Pointer-variable

Adresse der Pointer-variablen

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

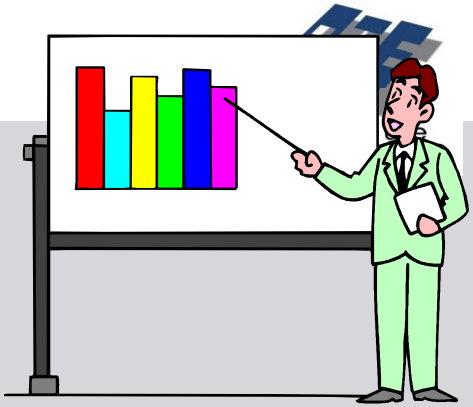
*) Speicher kann ein beliebiger Speicher sein, z. B. RAM, ROM, Register

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



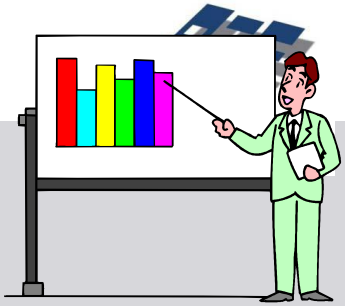
1. Pointer

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

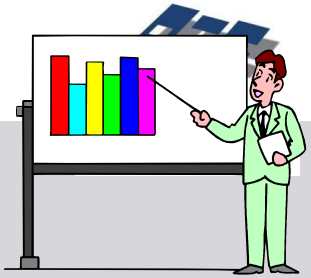


1. **Pointer**
   - **Pointer haben auf einer HW alle die gleiche Größe (z.B. 4 Bytes)**
   - **Pointer können auch auf Dateien (s. File In/Output) oder Funktionen zeigen**
   - **Pointer werden auch als *Referenz* bezeichnet**

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

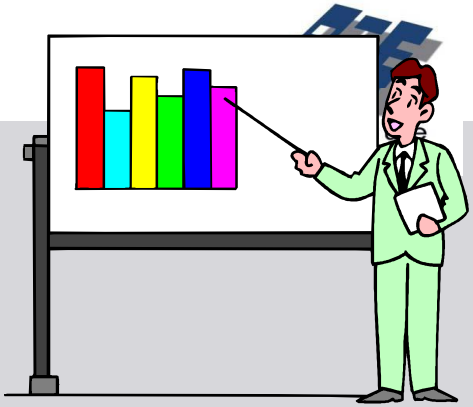


1. **Pointer**
   - **Deklaration Pointervariable in C: *Typ* Pointername***
   - *** drückt aus, dass es sich um einen Pointer handelt z.B.**
     - **int* intpointer; /* Pointer auf einen Integer */**
     - **char* charpointer /* Pointer auf Character */**
     - **int* register_X /* Pointer auf ein 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

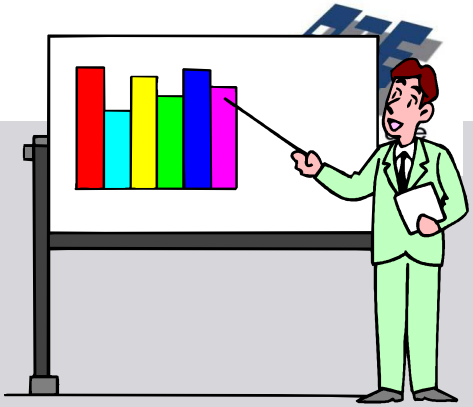
Adr.	Speicher*)
1	
2	
3	
4	
5	
...	
7153	„a“
...	
9999	
10000	

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

Pointer

1. **Pointer**
   - **Pointer als Parameter einer Funktion in C: ergtyp *f(typ* p, ...)***
   - *** 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

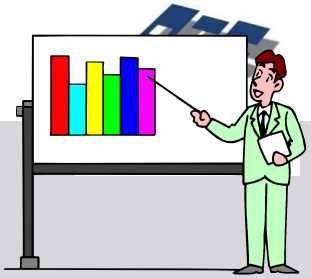
Pointer als Parameter einer Funktion

Funktion greift i.d.R. auf Inhalt zu

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

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

Pointer

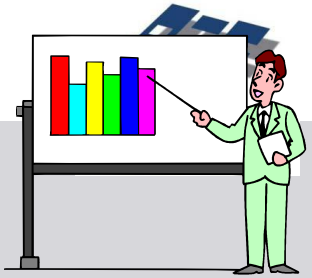


1. **Pointer**
   - **Pointer als Parameter und / oder Ergebnis einer Funktion**
     - **erspart Kopieren großer Datenmengen auf Parameter- oder Ergebnisposition**
     - **ist bei dynamischen Strukturen ohne Alternative**
     - **erfordert Vorsicht bei der Anwendung, da der Speicher, auf den der Pointer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

Pointer

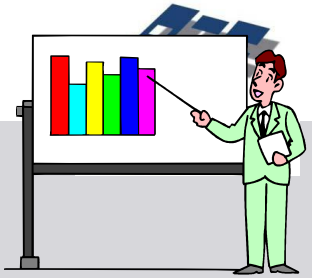
1. **Pointer**
   - **Pointervariable, Belegung mit einem Wert**
     - **intpointer = NULL; /* NULLPointer, zeigt nirgendwohin */**
     - **charpointer = 0xFF01; /* feste Adresse */**
     - **register_X = 0xFFAA /* feste Adresse */**
     - **Struktur *structpointer = &s; /* Adresse einer Variablen 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	

Pointer

1. Pointer

- **Pointervariable, Dereferenzieren**

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

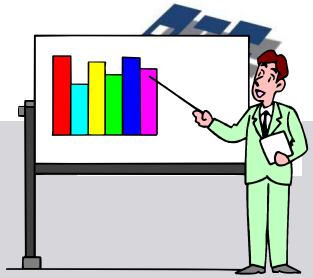
Wert

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

Pointer

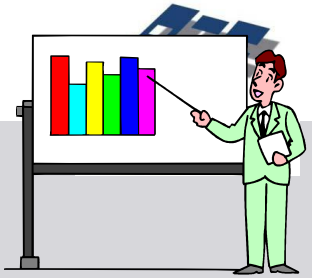
1. **Pointer**
   - **Pointervariable, Dereferenzieren**
     - ***charpointer = ´c`;**
     - ***register_X = 0xAA /* Register beschreiben */**
     - **f1 (*structpointer) /* Parameter */**
     - **x = *intpointer1 + *intpointer2; /* Addition der Werte, auf die intpointer1 und intpointer2 zeigen */**
     - **... 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

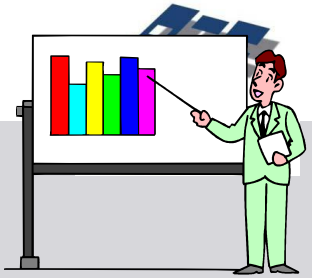


1. **Pointer**
   - **Pointervariable, Dereferenzieren**
     - **Pointer auf Strukturen**

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

   - **2 Möglichkeiten des Komponentenzugriffs bei Dereferenzierung**
     1. (*cpointer).real = 5.5;
     2. cpointer->real = 5.5;

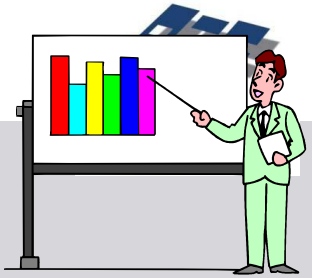
Pointer



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

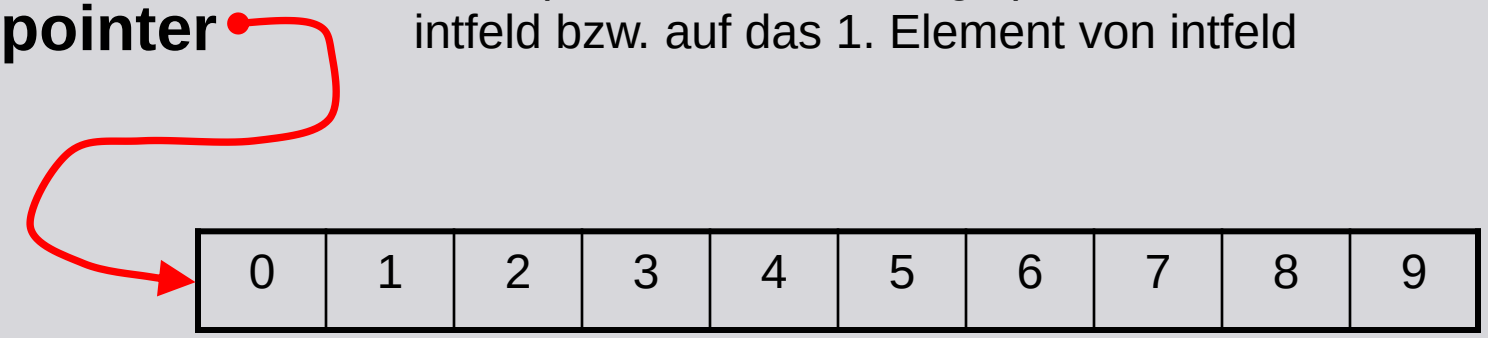
Pointer



2. Pointer und Arrays

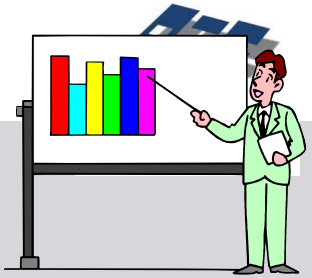
ipointer

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



intfeld

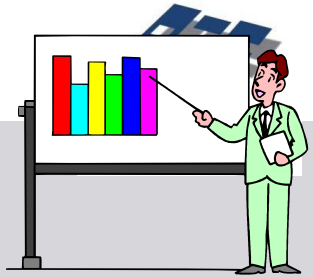
Pointer



2. Pointer und Arrays

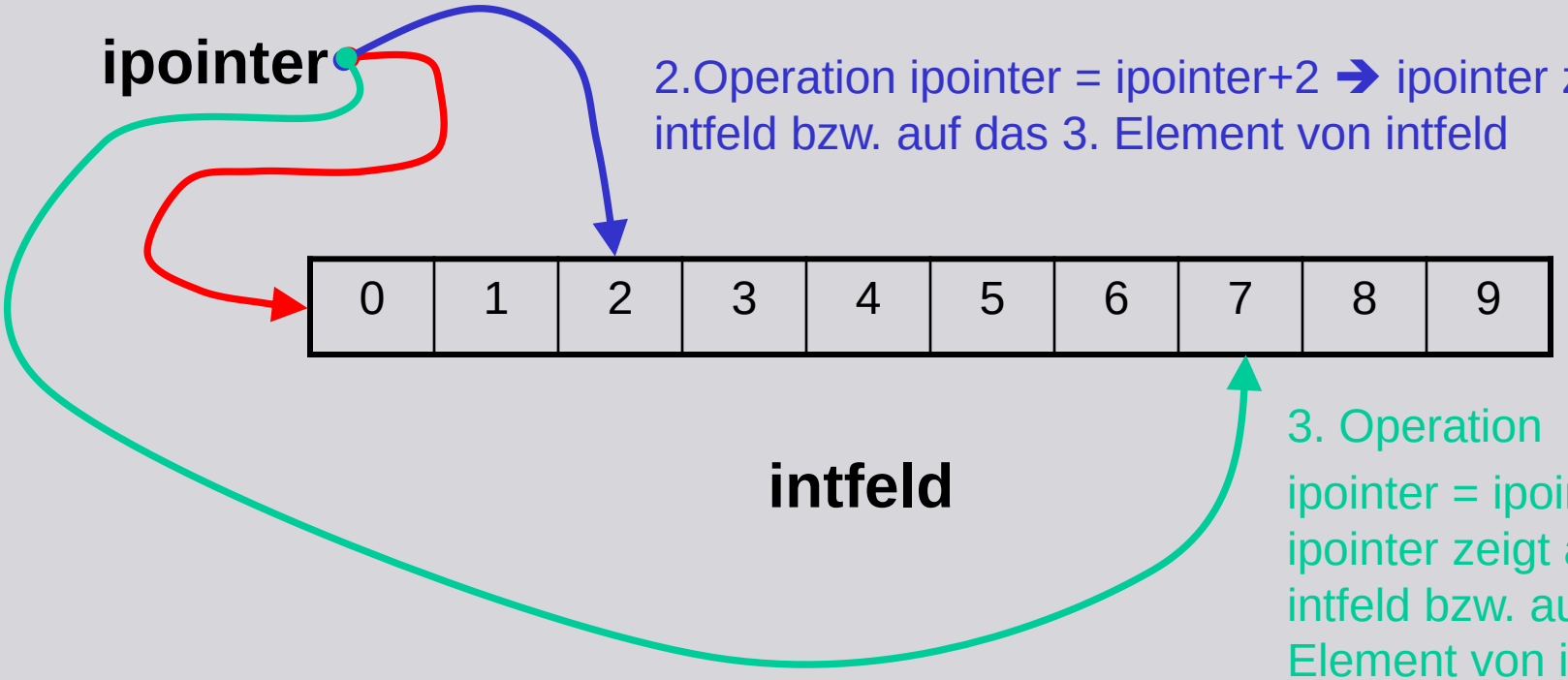
- durch Pointerarithmetik kann auf über ipointer auf die Elemente von intfeld zugegriffen werden.
- ipointer+1 zeigt auf das 2. Element von intfeld
- ipointer+2 zeigt auf das 3. Element von intfeld
- ipointer + k zeigt auf das (k+1)-te Element von intfeld
- dies gilt für alle Arraytypen, unabhängig vom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

Pointer



2. Pointer und Vektoren (Arrays)

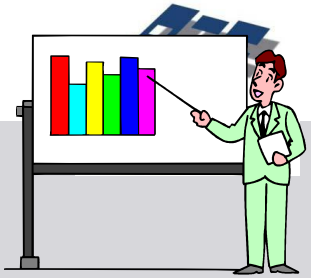
1,Operation ipointer = intfeld ➔ ipointer zeigt auf intfeld bzw. auf das 1. Element von intfeld



2.Operation ipointer = ipointer+2 ➔ ipointer zeigt auf intfeld bzw. auf das 3. Element von intfeld

3. Operation
ipointer = ipointer+5 ➔ ipointer zeigt auf intfeld bzw. auf das 8. Element von intfeld

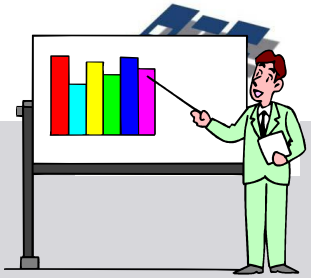
Pointer



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

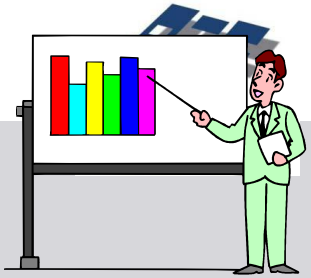


3. Pointer und Weiteres

- **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



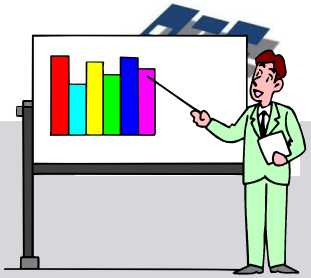
3. Pointer und Weiteres

- **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

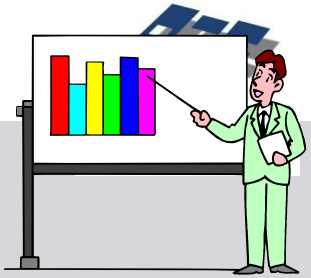


3. Pointer und Weiteres

- **Pointer können, wie Variable von normalen Typen, gecastet werden:**
  int *intpointer;
  unsinged int natzahl = 5;
  (unsigned int*) intpointer = &natzahl;

- **Analog zum Casten von Variablen muss Pointercasten vorsichtig durchgeführt werden!**

Pointer

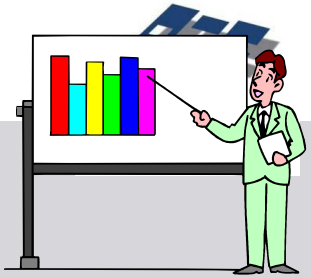


4. Pointer und Dynamische Speicherverwaltung

- 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)

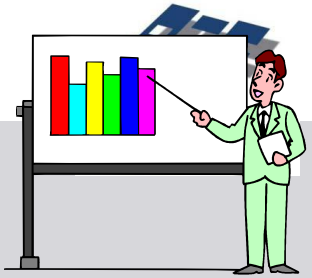
➔ Möglichkeit der expliziten dynamischen Speicherverwaltung durch Programmierer

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;
```

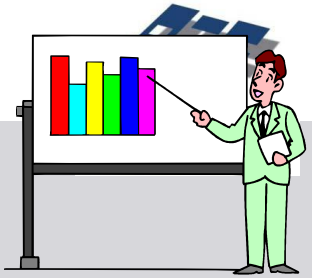
Variable j wird angelegt

j

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)

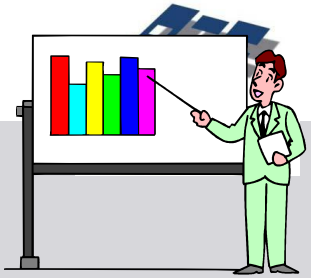
```
int j = 7;
...
{ int i = 5;
  ...
}
j = 10;
```

Variable i wird angelegt

	Adr.	Arb.-Speicher
	1	
	2	
	3	
	...	
	...	
	...	
	...	
i	9998	5
j	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;
```

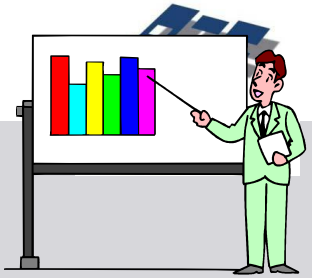
Variable i wird gelöscht, Stack freigegeben

j

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

Stack „schrumpft“

Pointer



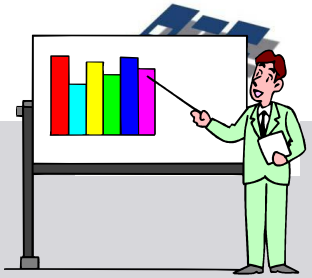
4. Pointer und Dynamische Speicherverwaltung

- 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“

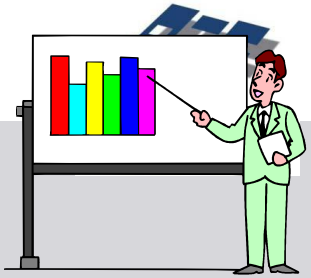
Adr.	Arb.-Speicher
1	
2	
3	
...	
...	
...	
...	
9998	
9999	
10000	

Grenze des belegten Heap („pulsiert“)

Stack wächst von „unten nach oben“

Grenze des belegten Stack („pulsiert“)

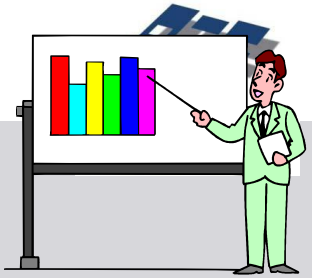
Pointer



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**

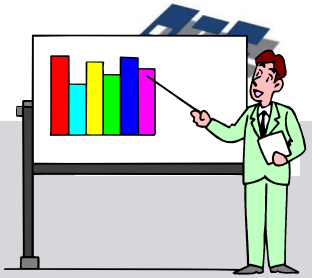
intptr = (int*) malloc(Byteanzahl);

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));

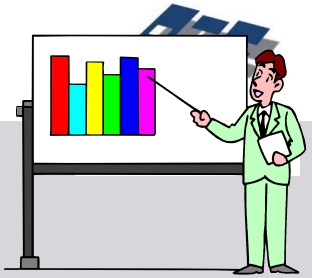
Cast auf richtigen Pointertyp

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

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

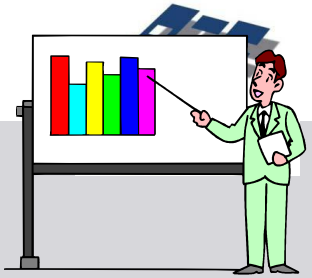
malloc(sizeof(int));
liefert Adresse
3555 zurück

Grenze des
belegten Stack

Grenze des
belegten Heap
vor malloc(...)

Grenze des
belegten Heap
nach malloc(...)

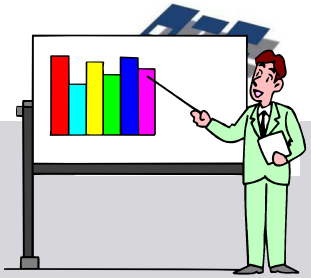
Pointer



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!

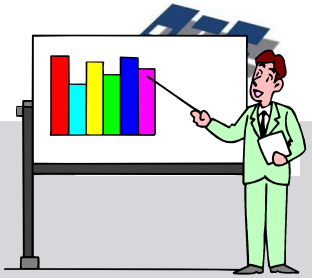
Pointer



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

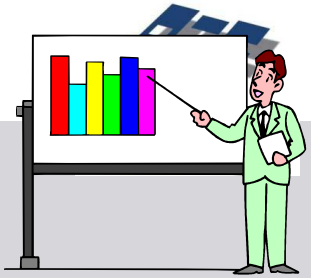
Pointer



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

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

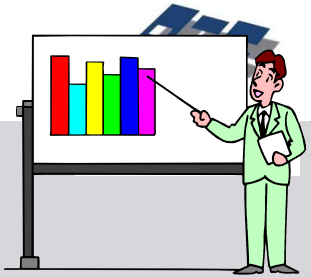
intptr

free(intptr) gibt belegten Speicherplatz wieder frei

Grenze des belegten Stack

Grenze des belegten Heap

Pointer

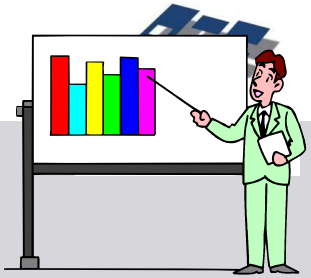


4. Pointer und Dynamische Speicherverwaltung

- 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

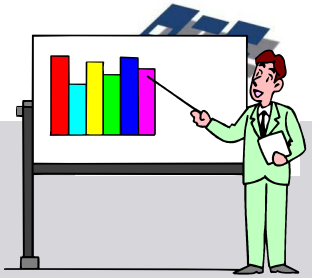


4. Pointer und Dynamische Speicherverwaltung

- **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

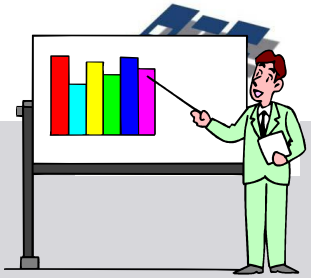


4. Pointer und Dynamische Speicherverwaltung

- 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

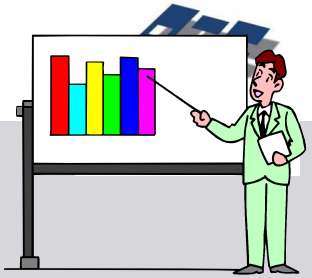


4. Pointer und Dynamische Speicherverwaltung

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

```
...
int* intptr = new int;
...
free (intptr); /* verboten */
```

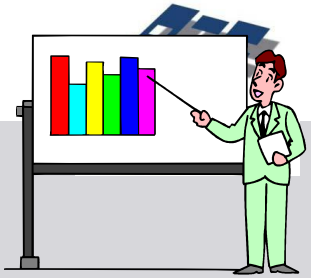
Pointer



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 letzlich 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

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??

Funktionen

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??

Funktionspointer

5. Funktionspointer

- 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 `*`

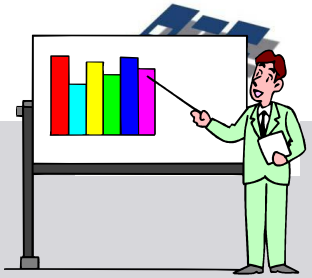
Funktionspointer

6. Funktionspointer

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

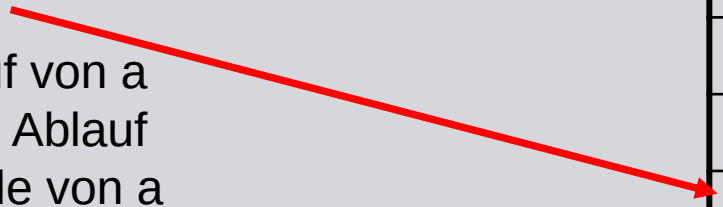
```
int a (int z)
{ if (z <0)
    return –z;
  else return z;
};

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



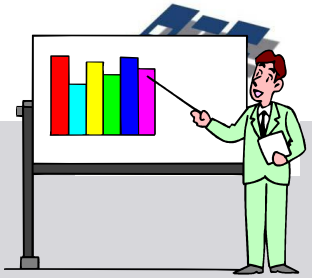
6. Funktionspointer

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



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

Funktionspointer



6. Funktionspointer

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.	Programm-speicher
1	
2	
3	
...	
3555	Int a (int z)
...	{ ...
...	...
3700	... }
...	
10000	

Funktionspointer

Hochschule Deggendorf

Zum Schluss dieses Abschnitts …

Noch Fragen ??