Fachhochschule Hannover
University of Applied Sciences and Arts

Bachelor Projekt

C-Programmierung WS 07/08

Betreuender Professor:

Prof. Dr.-Ing. Karl-Heinz Niemann
Fachbereich Elektro- und
Informationstechnik
Prozessinformatik / Automatisierungstechnik

Studenten:

Uwe Janssen, Fakultät I,
Mat. Nr.: 1033175
uwe.janssen@stud.fh-hannover.de

Christian Ziegner, Fakultät I,
Mat. Nr.: 1033861,
christian.ziegner@stud.fh-hannover.de

Aufgabenstellung:

- Die Programmiersprache C erlaubt die Verwendung von Zeigern (engl. Pointer).
- Im Rahmen dieses Projektes soll eine Lerneinheit von der Projektgruppe erstellt werden, die die Verwendung von Zeigern anschaulich für Studierende erklärt.
- Insbesondere soll eingegangen werden auf:

- Zeiger allgemein
- Zugriff mit Zeigern auf Felder (Arrays)
- Nutzung von Zeigern zur Übergabe von Parametern an Funktionen.

Die Ausführung des Lernprojektes ist von der Gruppe wählbar. Möglich wären z. B.:

- Interaktive Web-Applikation.
- Animierte FLASH-Applikation.

Text mit zusätzlichen Programmbeispielen.

Inhaltsverzeichnis

1. Einführung

Diese Webseite ist aus einem Semesterprojekt entstanden, dessen Aufgabe es war eine Lerneinheit zu erstellen, die die Verwendung von Zeigern (engl. Pointer, im Folgenden wird ausschließlich die englische Bezeichnung „Pointer“ verwendet) anschaulich für Studierende erklärt. Da sich diese Lerneinheit nur mit dem Thema Pointer beschäftigt ist es sinnvoll gewisse Vorkenntnisse in der Programmiersprache C mitzubringen. So sollten der grundsätzliche Aufbau eines C-Programms bekannt und Begriffe wie Compiler, Datentypen, Operatoren, Schleifen, Felder und Funktionen nicht fremd sein. Viele Studierende tun sich schwer damit die Funktion von Pointern zu verstehen, doch ist das Grundprinzip einmal verstanden, so sollte es eine Leichtigkeit sein mit Pointern umzugehen. Wir haben uns bemüht in dieser Lerneinheit das Thema möglichst anschaulich zu vermitteln, damit der lernende sich gut vorstellen kann was im Detail abläuft.

Die Lerneinheit ist folgendermaßen aufgebaut:

Es gibt insgesamt drei Einheiten. Im ersten Teil sollen Grundlagen vermittelt werden die verständlich machen, warum es Pointer gibt und wie Pointer in einem C-Programm eingesetzt werden. Im zweiten Teil wird der besondere Einsatz von Pointern im Bezug auf Felder erklärt und im dritten Teil werden Pointer im Zusammenspiel mit Funktionen behandelt. Nach jeder Einheit folgen ein Fragenkatalog in dem zuvor behandelte Inhalte in Form von Single Choice abgefragt werden und ein Aufgabenteil in dem das erlernte praktisch angewendet und vertieft werden soll.

2. Pointer Grundlagen

Um zu verstehen was ein Pointer macht, ist es hilfreich sich die Bedeutung des Wortes klar zu machen. Ein Pointer, im deutschen ein Zeiger, ist ein Symbol das auf etwas zeigt, z.B. ein Pfeil im Straßenverkehr der einem die Richtung weist oder der Zeiger einer Uhr der auf die Uhrzeit zeigt. Also hat ein Pointer eine „auf etwas zeigen“ Funktion. Worauf kann oder sollte aber in einem Programm gezeigt werden? Wenn ein C-Programm geschrieben wird, dann werden ganz selbstverständlich Variablen verwendet. Für die Variable werden ein Datentyp und ein Name festgelegt. Außerdem kann ein Wert zugewiesen werden, was folgendes Beispiel zeigt.

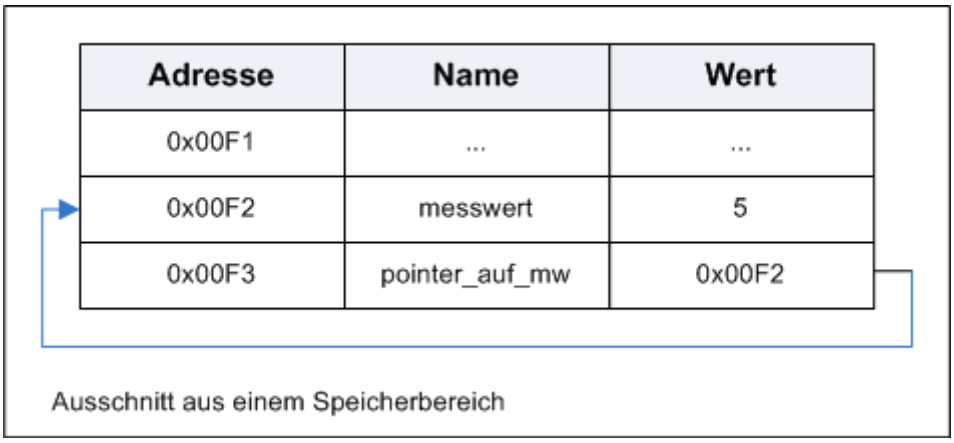
```
int messwert = 5;
(Datentyp Variablenname = Wert)
```

Beim Ausführen des Programms wird dieser Wert incl. Name und Datentyp im Speicher des Computers abgelegt. Der Anwender bekommt dabei nicht mit, dass dieser Datensatz intern einen zusätzlichen Hexadezimalen Wert, die Speicheradresse, vom Compiler zugewiesen bekommt. Dieser ist notwendig um bei der Vielzahl an möglichen Speicherplätzen den richtigen Speicherplatz für einen Wert wieder zu finden. (Der Compiler bekommt vom System einen bestimmten Speicherbereich zugewiesen, aus dem er wiederum willkürlich den Variablen Speicheradressen zuweisen kann.) Somit könnte die Speicherstelle für die obige Deklaration folgendermaßen aussehen:

Adresse	Name	Wert
0x00F1	...	...
0x00F2	messwert	5
0x00F3	...	...

Ausschnitt aus einem Speicherbereich

Der gespeicherte Wert kann nun über den Namen als auch über die Adresse abgerufen oder geändert werden. Wobei das Abrufen eines Wertes über die direkte Adresse eher unüblich ist, da sie zu jeder Programmlaufzeit willkürlich zugewiesen wird und in hexadezimaler Schreibweise nicht leicht zu merken ist. Jedoch ist ein indirekter Zugriff möglich der in verschiedensten Programmen Anwendung findet. Dieser wird über den Pointer ermöglicht, der einmal deklariert und initialisiert, auf die Adresse des Speicherplatzes „zeigt“ in dem der Wert einer Variablen abgelegt ist. Folgender Ausschnitt aus einem Speicherbereich verdeutlicht dies:



Der Wert des Pointers ist, wie im Beispiel erkennbar, die Adresse der Variable auf die der Pointer „zeigt“. Somit kann ein Pointer als normale Variable angesehen werden die als Wert, anstelle von beliebigen Daten, Speicheradressen enthält. Um eine Pointervariable zu deklarieren muss bekannt sein welchen Datentyp die Variable, auf die der Pointer zeigen soll, hat. Diese Angabe ist wichtig für die Sichtweise des Pointers. Sie gibt an wie viele Bytes ab der Anfangsadresse (z.B. bei einer Ausgabe der Variable) ausgewertet werden müssen. Außerdem wird ein so genannter Inhaltsoperator < * > verwendet, der den Compiler erkennen lässt das die Variable ein Pointer ist.

```
int *pointer_auf_mw;
(Datentyp auf den der Pointer zeigt – Inhaltsoperator - Pointername)
```

Zur Initialisierung des Pointers werden der Name der Variable, auf die gezeigt werden soll, und der Adressoperator < & > (er gibt die Anfangsadresse einer Variablen zurück) benötigt.

```
pointer_auf_mw = &messwert;
(Pointername = Adressoperator Variablenname)
```

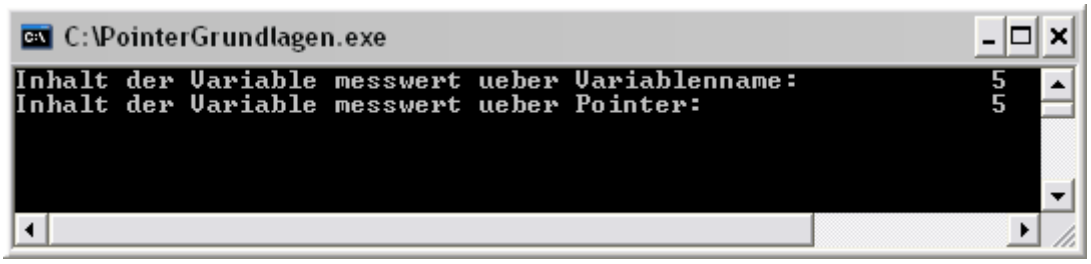
Wenn die Variable, auf die der Pointer zeigen soll, noch nicht bekannt ist muss dieser mit NULL initialisiert werden. Der Pointer wird sozusagen geerdet was ihn auf einen definierten Zustand setzt. Dadurch wird vermieden, dass der Pointer ungültige Daten enthält.

```
Pointer_auf_mw = NULL;
```

Über vorherige Befehlsfolge wurde als Wert für den Pointer die Adresse der Variable messwert im Speicher abgelegt. Folgende Befehlszeilen zeigen auf, welche Möglichkeiten es gibt den Wert der Variable messwert auszugeben. Der Inhaltsoperator kann hier folgendermaßen verstanden werden: „Inhalt der Adresse auf den der Pointer zeigt“.

```
printf("Inhalt der Variable messwert über Variablenname:%d", messwert);
printf("Inhalt der Variable messwert über Pointer:%d", *pointer_auf_mw);
```

Folgende Zeilen erscheinen in der Bildschirmausgabe:



In beiden Fällen wird der Inhalt der Variable messwert ausgegeben. Mit folgenden Befehlszeilen kann überprüft werden, ob die Adresse der Variable und der Wert des Pointers tatsächlich gleich sind.

```
printf("Adresse der Variable messwert:%d", &messwert);
printf("Inhalt der Variable pointer_auf_mw:%d", pointer_auf_mw);
```

Und tatsächlich belegt das Ergebnis, wie folgende Bildschirmausgabe beweist, dass die Adressen gleich sind. Dies muss auch so sein, da die Adresse der Variable messwert als Wert in der Pointervariable gespeichert ist.

Der Wert der Variable messwert kann entweder direkt oder auch über einen Pointerzugriff geändert werden:

```
messwert = 7;                    /* direkte Änderung */
*pointer_auf_mw = 7;             /* Änderung über Pointerzugriff */
```

Grundsätzlich belegt eine Pointervariable 4 Byte im Speicher (bei einem 32 Bit Adressbus, bei einem 64 Bit Adressbus sind es 8 Byte). Ob ein Pointer eine gültige Adresse enthält wird vom Compiler nicht überprüft. Deshalb kann es nur im Sinne des Programmierers sein, dieses selbständig zu kontrollieren. Um im Variablennamen kenntlich zu machen das es sich bei der Variable um einen Pointer handelt, können verschiedene Möglichkeiten der Namensgebung genutzt werden: `pointer_auf_mw; pt_messwert; messwert_pt`. Grundsätzlich ist dem Programmierer die Gestaltung des Pointer-Variablennamen freigestellt, jedoch sollte er im Programm einheitlich sein und anderen Programmierern kenntlich machen, dass eben diese Variable ein Pointer ist. Welche Vorteile die Verwendung von Pointern bei der C-Programmierung noch haben kann, wird in den nachfolgenden Einheiten verdeutlicht.

2.1 Fragen

1. Pointer speichern Anfangsadressen mit 8 Byte (bei einem 32 Bit Datenbus). (falsch)
2. Pointer können nur für `int`-Variablen erstellt werden. (falsch)
3. Die Anfangsadresse einer Variablen erhält man mit dem `&`-Operator. (wahr)
4. `*int_pt == 5;` weist der Speicherzelle auf die `int_pt` zeigt den Wert 5 zu. (falsch)
5. Der Compiler überprüft beim kompilieren, ob der Pointer auf gültige Daten zeigt. (falsch)
6. Mit dem *-Operator wird die Adresse des Pointers zurückgegeben. (falsch)
7. `double_pt1 == double_pt2;` prüft, ob die Zeiger auf die gleiche Adresse zeigen. (wahr)
8. `float_pt = NULL;` bedeutet das der Pointer auf keine Speicheradresse zeigt. (wahr)
9. `*short_pt = 0xCB977C33;` weist dem Pointer direkt eine Speicheradresse zu. (falsch)
10. Aus dem Datentyp eines Pointers erkennt man wie viele Bytes gelesen werden. (wahr)

2.2 Aufgaben

2.2.1 Pointer deklarieren und initialisieren

Schreibe ein Programm mit drei Variablen. Deklariere für jede Variable einen Pointer und initialisiere ihn mit der jeweiligen Anfangsadresse. Erzeuge zusätzlich einen Pointer der auf keine Variable zeigt.

2.2.2 Adresse ausgeben

Ergänze das Programm der ersten Aufgabe um eine Ausgabe der Speicheradressen aller drei Variablen. Nutze den Adressoperator sowie den Pointer dafür und vergleiche diese Werte miteinander. Welche Ausgabe hat der Pointer, der auf keine Variable zeigt?

2.2.3 Verändern über Pointerzugriff

Ergänze das Programm aus Aufgabe 2.2.1. Weise jeder Variablen über den Pointer einen neuen Wert zu und gib ihn über den Variablennamen aus. Dann ändere die Werte über den Variablennamen und gib sie diesmal über den Pointer aus.

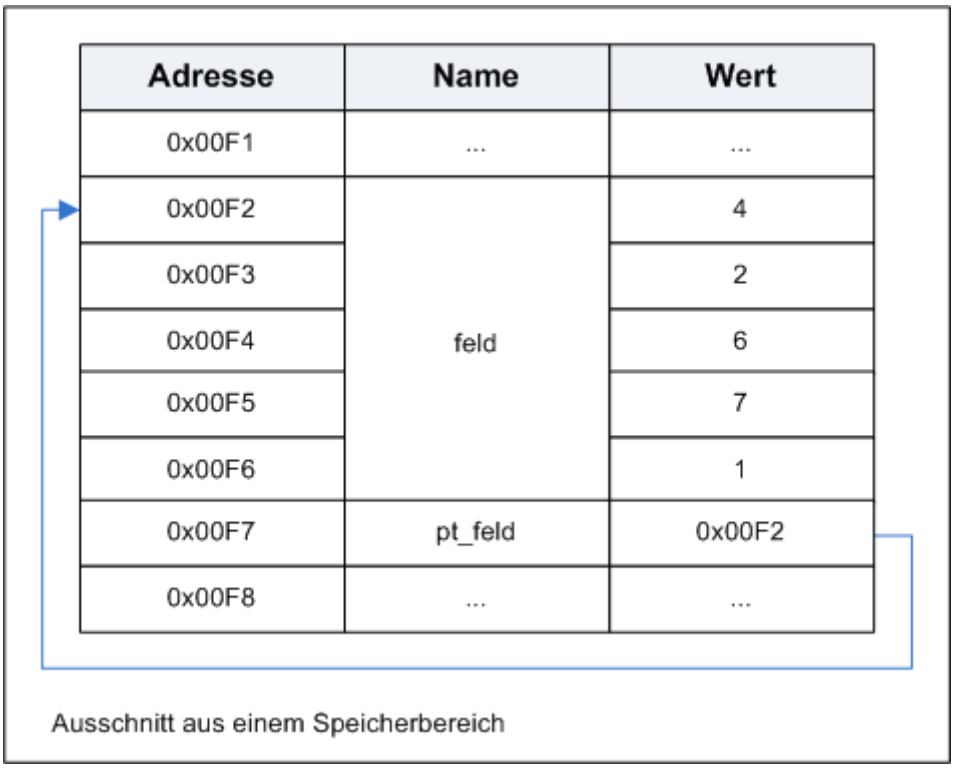
3. Pointer & Felder

3.1 Pointer mit eindimensionalen Feldern

Eine große Bedeutung haben Pointer auch im Zusammenspiel mit Feldern / Arrays. Die Anwendung ist ähnlich wie bei den, in der ersten Einheit behandelten, Variablen. Die Deklaration und Initialisierung von Feld und Pointer sieht folgendermaßen aus.

```
int feld[5] = {4, 2, 6, 7, 1};   /* int-Feld deklarieren und initialisieren*/
int *pt_feld;                    /* Pointer deklarieren */
pt_feld = &feld[0];              /* Pointer auf Anfang des Feldes setzen */
```

Wie die Speicherzuordnung dafür aussehen könnte, zeigt folgendes Bild:



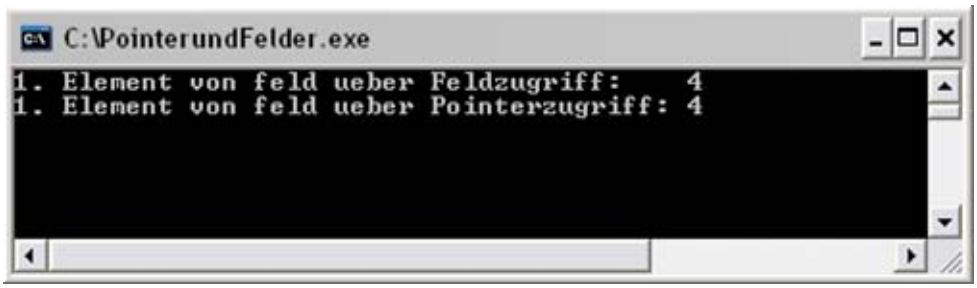
Eine weitere Möglichkeit den Pointer auf den Anfang des Feldes zu setzten wäre

```
pt_feld = feld;          /* entspricht pt_feld = &feld[0]) */
```

da der Feldname ohne Index die Anfangsadresse des ersten Feldelements ist. Zu Beachten ist, dass jedes deklarierte Feld mit dem Feldindex „0“ beginnt. Um auf die Elemente des Feldes zuzugreifen gibt es wieder zwei Möglichkeiten. Entweder der direkte Feldzugriff über den Index oder indirekt über den Pointer. Wie der Zugriff auf das erste Feldelement aussehen kann zeigt folgendes Beispiel:

```
printf("1. Element von feld über Feldzugriff: %d", feld[0]);
printf("1. Element von feld über Pointerzugriff: %d", *pt_feld);
```

Bei der ersten Möglichkeit wird der Inhalt des ersten Feldelements zurückgegeben. Die zweite Möglichkeit gibt den Inhalt der Adresse wieder, auf die der Pointer zeigt, welche in diesem Fall ebenfalls die Adresse des ersten Feldelements ist. Die Bildschirmausgabe belegt, dass bei zwei unterschiedlichen Methoden der selbe Wert zurückgegeben wird.



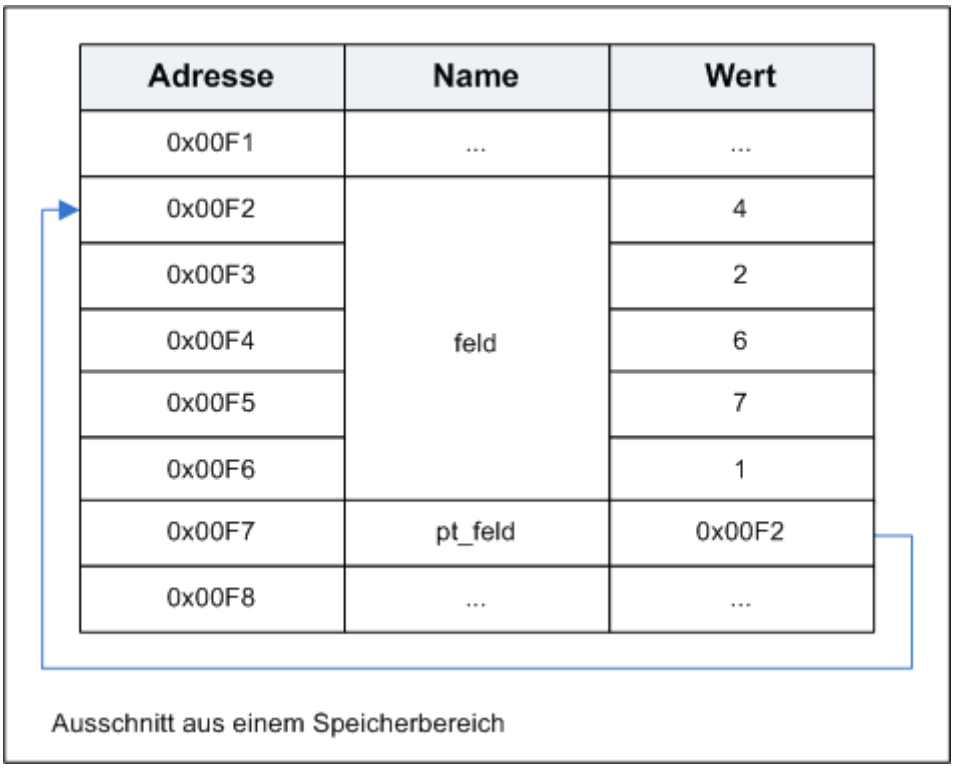
Um auf das nächste Element des Feldes anzusprechen wird beim Feldzugriff über den Variablennamen einfach der Index erhöht.

```
printf("3. Element von feld über Feldzugriff: %d", feld[2]);
```

Wie aber ist dies über den Pointer möglich, der ja nur auf die Anfangsadresse des Feldes zeigt? Die erste Möglichkeit ist, den Pointer durch das Feld „wandern“ zu lassen. Dabei wird die Adresse, auf die der Pointer zeigt, dem Zugriff angepasst und jeweils um den entsprechenden Wert erhöht (Diese Methode ist auch sehr gut für Schleifen geeigent). Wie folgendes Beispiel zeigt wird der Inhalt des Pointers um die entsprechende Anzahl erhöht.

```
pt_feld = pt_feld + 2;
printf("3. Element von feld über Pointerzugriff: %d", *pt_feld);
```

Da dem Compiler der Datentyp bekannt ist (bei der Deklaration festgelegt) weiß er wie viele Bytes im Speicher er weiter „wandern“ muss um das nächste oder übernächste Feldelement zu adressieren. Wie das "Wandern" des Pointers im Speicher aussehen kann, zeigt folgende Animation:

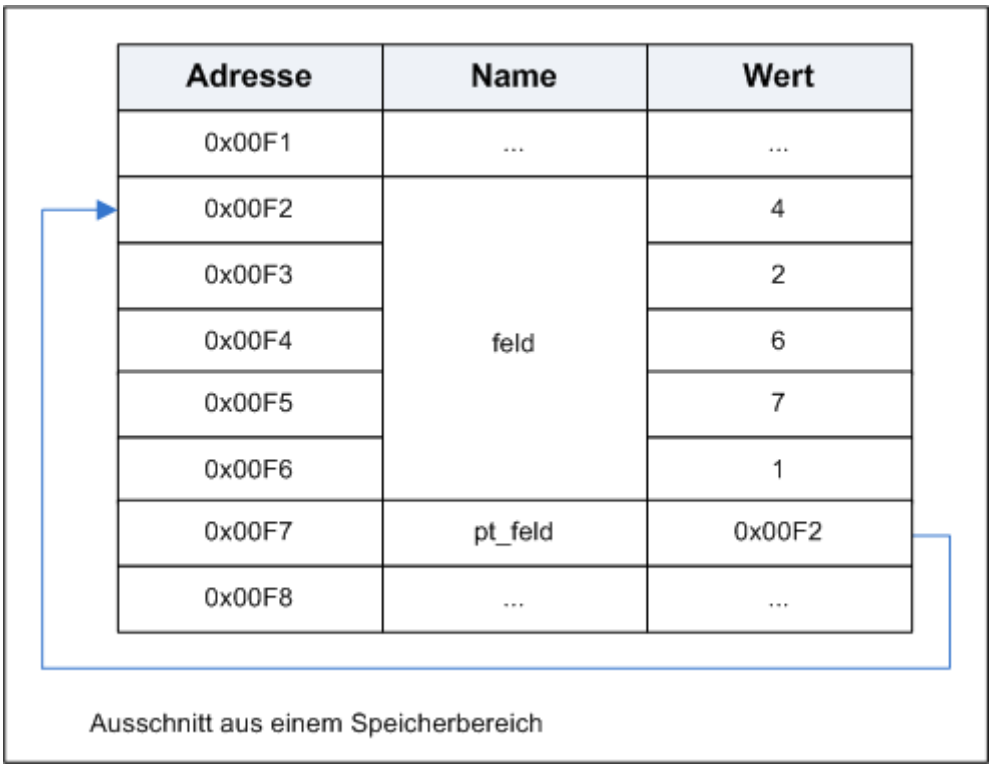


Natürlich kann der Pointer auch kreuz und quer durch das Feld wandern. Dabei sollte unbedingt darauf geachtet werden die Feldgrenzen einzuhalten und den Pointer evtl. vor einer Operation auf die Anfangsadresse des Feldes zurück zu setzen.

Eine weitere Möglichkeit ist, den Pointer zu „verbiegen“. Dabei behält der Pointer die Anfangsadresse bei und der so genannte Offset wird um die entsprechende Anzahl an Feldelementen erhöht. Angewendet wird diese Möglichkeit folgendermaßen:

```
printf("3. Element von feld über Pointerzugriff: %d", *(pt_feld+2));
```

Auch hier ist es wieder der bekannte Datentyp der angibt, um wie viele Bytes der Pointer „verbogen“ werden muss, um ein bestimmtes Feldelement zu adressieren. Der Vorteil dieser Methode ist, das der Pointer immer seine Anfangsadresse behält und somit nicht darauf geachtet werden muss, diese zurück zu setzten. Das "Verbiegen" des Pointers kann im Speicher folgendermaßen aussehen:



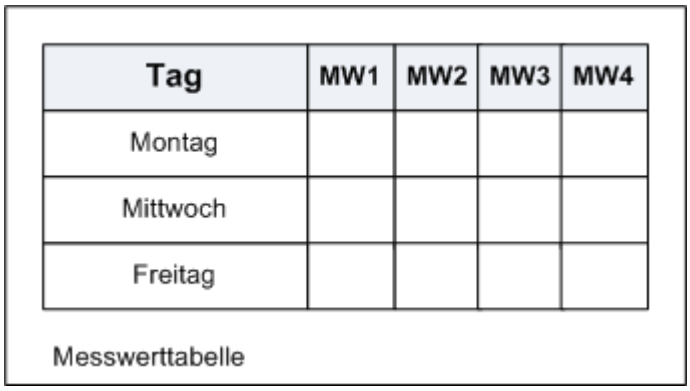
Hier kann der Pointer ebenfalls über entprechende Operationen kreuz und quer „verbogen“ werden. Wichtig ist auch hier die Einhaltung der Feldgrenzen. Die Möglichkeiten des „verschiebens“ und „verbiegens“ des Pointers werden häufig als Pointerarithmetik bezeichnet.

Speziell bei eindimensionalen Feldern ist auch ein Pointerzugriff über den Index möglich. Wie dieser aussehen kann zeigt folgendes Beispiel:

```
printf("1. Element von feld über Pointerzugriff mit Index: %d", pt_feld[0]);
```

3.2 Pointer mit mehrdimensionalen Feldern

Wie Pointer auf mehrdimensionale Felder angewendet werden, wird am Beispiel eines zweidimensionalen Feldes erklärt. Angenommen es werden an 3 Tagen jeweils 4 Messwerte z.B. die Temperatur aufgenommen, so können diese in einer Tabelle dargestellt werden, in der die Tage die Zeilen und die Messwerte die Spalten sind.



Tag	MW1	MW2	MW3	MW4
Montag				
Mittwoch				
Freitag				

Messwerttabelle

Ein für diese Tabelle erstelltes Feld kann folgendermaßen deklariert werden.

```
int messwerte [3][4];          /* 3 Zeilen, 4 Spalten */
```

Um die Elemente des Feldes nun mit Werten zu füllen, werden üblicherweise Schleifenstrukturen benutzt die Spalte für Spalte und Zeile für Zeile ablaufen und einen Wert ablegen. Der Programmcode zeigt ein Beispiel bei dem jedes Feldelement mit einer 0 initialisiert wird:

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int messwerte [3][4];
06     int i, j = 0;
07
08     /* Initalisierung der Feldelemente mit 0 */
09     for (i=0; i<3; i++)
10         {
11          for (j=0; j<4; j++)
12              {
13                 messwerte [i][j] = 0;
14               }
15          }
16     /* Ausgabe der Feldelemente */
17     for (i=0; i<3; i++)
18         {
19          for (j=0; j<4; j++)
20              {
21               printf ("Tag:%d MW:%2d: %d\n", i+1, j+1, messwerte [i][j]);
22               }
23          }
24     fflush(stdin);
25     getchar();
26     return 0;
27 }
```

Im Speicher ergibt sich für das angelegte Feld `messwerte` folgendes Bild:

Adresse	Name	Wert	Zeile / Spalte
0x00F1	...	...	...
0x00F2	messwerte	0	0/0
0x00F3		0	0/1
0x00F4		0	0/2
0x00F5		0	0/3
0x00F6		0	1/0
0x00F7		0	1/1
0x00F8		0	1/2
0x00F9		0	1/3
0x0101		0	2/0
0x0102		0	2/1
0x0103		0	2/2
0x0104		0	2/3
0x0105	...	...	...

Ausschnitt aus dem Speicherbereich

Wie an den unterschiedlichen Farben zu erkennen ist, werden die einzelnen Zeilen der Tabelle nacheinander im Speicher abgelegt. Diese Eigenschaft hat für den Pointer eine sehr große Bedeutung, da dieser den Zugriff auf das zweidimensionale Feld sehr stark vereinfacht. Der Pointer wird mit der Anfangsadresse des Feldes initialisiert und durch „verschieben“ oder „verbiegen“ des Pointers kann dieser vom Anfang bis zum Ende des Feldes „wandern“ um z.B. die Werte der Elemente auszugeben oder sie zu initialisieren. Folgender Programmcode zeigt ein Beispiel bei dem jedes Feld über einen sich „verbiegenden“ Pointer mit 0 initialisiert wird:

```
01 #include <stdio.h>
02
03 int main(void)
04 {
05     int messwerte [3][4];
06     int i, j = 0;
07     int *pt_messwerte = &messwerte [0][0];
08
09     /* Initalisierung der Feldelemente mit 0 */
10     for (i=0; i< 3*4; i++)
11         {
12             *(pt_messwerte + i)=0;
13         }
14     /* Ausgabe der Feldelemente */
15     for (i=0; i< 3*4; i++)
16         {
17             printf ("Messwert -> %2d: %d \n", i+1, *(pt_messwerte + i));
```

```
18          }
19      fflush(stdin);
20      getchar();
21      return 0;
22       }
```

Ebenso kann auch der sich „verschiebende“ Pointer angewendet werden, wobei zu beachten ist, dass dieser vor jeder Operation auf den Feldanfang zurückgesetzt werden muss. Analog zu den Beispielen mit zweidimensionalen Feldern können Pointer auch sehr einfach bei Feldern mit mehr als zwei Dimensionen angewendet werden, da die Elemente jeder weiteren Dimension im Speicherbild an die vorhandenen Elemente angehängt werden.

3.3 Fragen

1. Der Feldname ohne Index ist Anfangsadresse des ersten Feldelements. (wahr)
2. Bei einem Feld mit 5 Elementen beginnt der Index mit 1. (falsch)
3. mit `pt_feld++` in einer Schleife kann der Pointer durchs Feld „wandern“. (wahr)
4. `pt_feld[0][0]` ist eine gültige Adressierung eines Feldelements. (falsch)
5. Über `*(pt_feld +4)` wird das 5. Element des Feldes angesprochen. (wahr)
6. `pt_a = &Feld[5]` weist dem Pointer die Adresse des 6. Feldelements zu. (wahr)
7. Pointer können durch mehrdimensionale Felder wandern. (wahr)
8. Mehrdimensionale Felder werden im Speicher ungeordnet angelegt. (falsch)
9. Feldelemente über Pointer anzusprechen ist sehr kompliziert. (falsch)
10. Ein Pointer kann bei dreidimensionalen Feldern nicht „verbogen“ werden. (falsch)

3.4 Aufgaben

3.4.1 ASCII Tabelle

Schreibe ein Programm welches ein Feld mit den Werten der ASCII-Tabelle initialisiert (kleiner Tip: 255 Elemente, es gibt mehrere Möglichkeiten den Datentyp char auszugeben) und lass danach die ASCII-Zeichen untereinander auf dem Bilschirm ausgeben. Das ganze soll mit Pointerzugriff realisiert werden

3.4.2 Messwerttabelle

Schreibe ein Programm mit dem 10 Messwerte über einen Pointerzugriff in ein Feld eingelesen werden können und gebe anschließend die eingelesenen Werte nacheinander (natürlich über Pointerzugriff) wieder aus.

3.4.3 Schachbrett

Schreibe ein Programm mit dem die 64 Felder (8x8) eines Schachbretts über einen Pointer ihrer Farbe nach initialisiert werden (z.B. für schwarz = '#', für weiß = ' '). Gebe das Schachbrett anschließend formatiert (also in Schachbrettform) aus und überpüfe so, ob die Anordnung der einzelnen Felder korrekt ist.

4. Pointer & Funktionen

Beim Programmieren in C wird viel mit Funktionen gearbeitet, weil es den Quelltext übersichtlicher macht und es möglich ist, Teilprogramme von anderen Programmierern schreiben zu lassen. Dazu müssen nur die zu übergebenen Variablen und der Rückgabewert abgesprochen werden. Außerdem besteht die Möglichkeit Funktionen in anderen Programmen noch einmal zu benutzen. Es ist nicht zu vergessen, dass

```
void main(void){…}
```

auch eine Funktion ist! In diesem Fall werden keine Variablen an sie übergeben und sie liefert keinen Rückgabewert. (Die `main` - Funktion wird von nun an nicht jedes Mal extra als Funktion erwähnt.)

Vielleicht hat sich der eine oder andere Leser dieser Lerneinheit schon gefragt, warum es diese Pointer denn nun gibt, wenn man die Speicherinhalte mit den Variablennamen und über die Pointer ansprechen kann. Man kommt in C um Pointer nicht herum wenn,

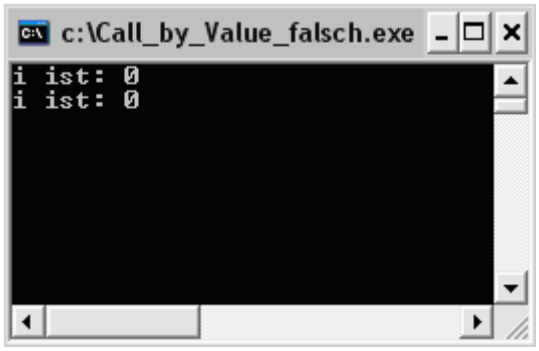
1. *man mit Funktionen arbeitet, die Speicherinhalte direkt manipulieren*
2. *man mehrere Rückgabewerte in einer Funktion hat*
3. *es um dynamische Speicherverwaltung geht*

Die ersten beiden Punkte werden an dieser Stelle anhand von Beispielen erläutert und der dritte Punkt wird erst im 2. Semester in Softwaretechnik behandelt.

4.1 Funktionen, die Speicherinhalte direkt manipulieren

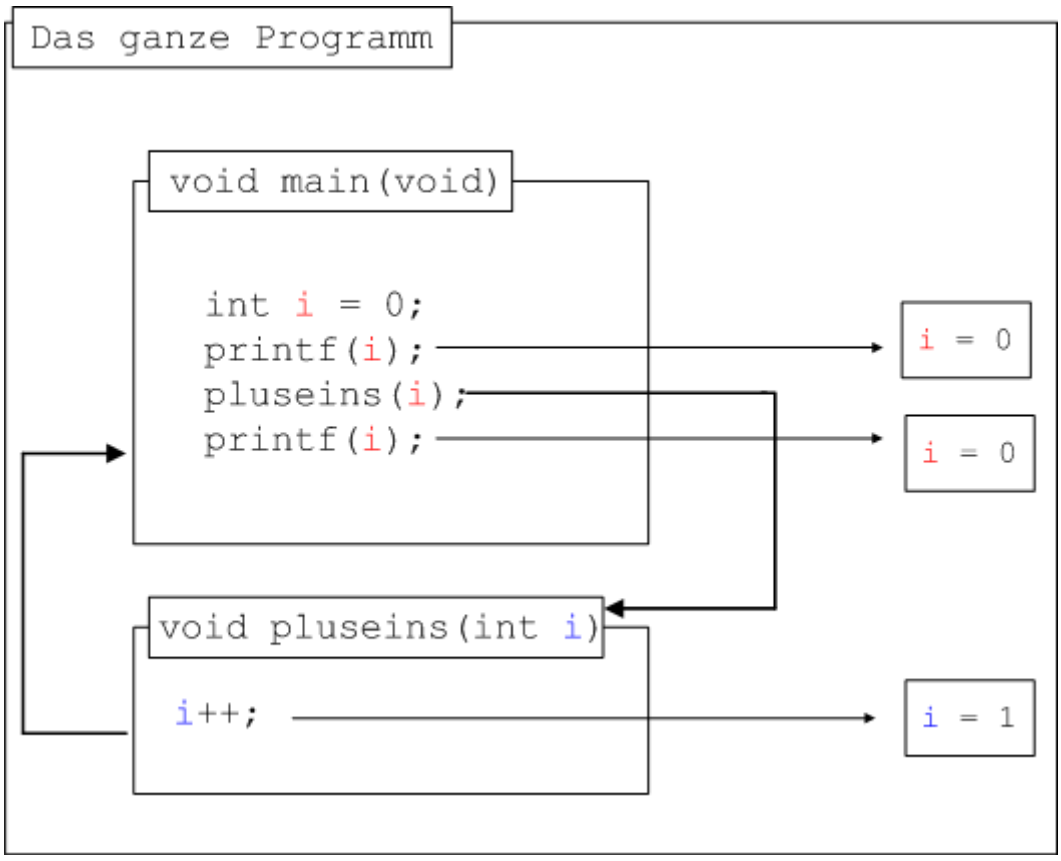
Wenn man mit Funktionen arbeitet, ist es wichtig, dass man sich den Gültigkeitsbereich der Variablen klar macht. Wenn man das nicht tut, kann es zu ungewollten Effekten kommen, was folgendes Beispiel (Quelltext 4.1) verdeutlicht. Es soll eine `int` Variable definiert und initialisiert werden und dann mit einer Funktion um eins erhöht werden. Um die Variable zu prüfen, soll eine Ausgabe auf dem Bildschirm vorgenommen werden.

```
01 #include <stdio.h>                  /*Header einbinden        */
02
03 void pluseins(int i);              /*Prototyp der Funktion   */
04 int main(void)                     /*Beginn, main - Funktion */
05 {
06     int i=0;                       /*dekl.und initialisieren */
08     printf("i ist: %d\n",i);       /*Testausgabe - Bildschirm*/
09     pluseins(i);                   /*Funktionsaufruf         */
10     printf("i ist: %d\n",i);       /*Testausgabe - Bildschirm*/
11     fflush(stdin);                 /*Tastaturpuffer löschen  */
12     getchar();                     /*auf Tastatur warten     */
13     return 0;                      /*0 zurückgeben an Konsole*/
14 }
15
16 void pluseins(int i)               /*Definition der Funktion */
17 {
18     i++;                           /*i um eins erhöhen       */
19 }
```



Es wurde beim Programmieren fälschlicherweise angenommen, dass es sich bei der Variable `i` im ganzen Quelltext um ein und die selbe Variable handelt. Das stimmt aber nicht. In Zeile 9 wird die Funktion aufgerufen und die Variable wird der Funktion „übergeben“. Diese Übergabe bedeutet aber nicht, dass sich die Variable `i` aus der `main` – Funktion nun in der `pluseins` – Funktion befindet. Es ist vielmehr so, dass die `pluseins` – Funktion sich den Wert `i` aus der `main` – Funktion

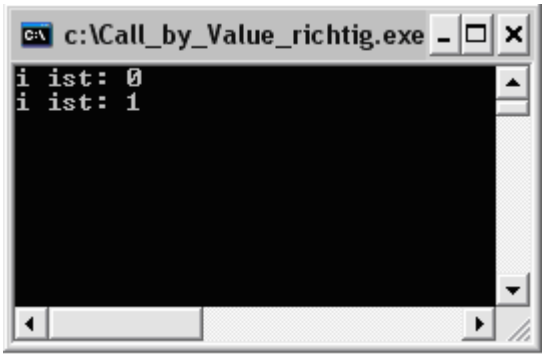
„anschaut und kopiert“ und dann mit einer eigenen i – Variable weiterarbeitet. Diese Art des Funktionsaufrufes wird Call-by-Value genannt. Man könnte auch sagen, dass die i – Variable der main – Funktion von der i – Variable der pluseins – Funktion überdeckt wird. In Bild 4.1 ist der Programmablauf einmal grafisch dargestellt.



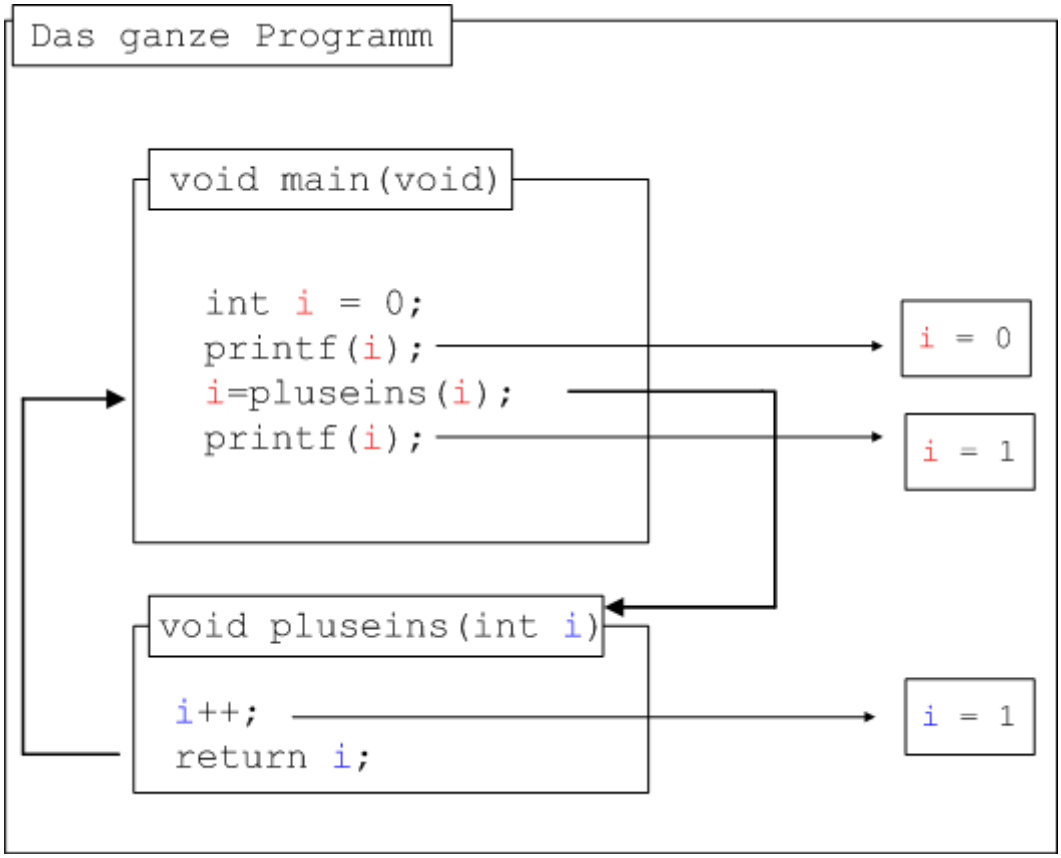
Damit das Programm tut, was es soll (eine 0 und dann eine 1 ausgeben), ist es nötig den Quelltext zu verändern. Die pluseins – Funktion wird so verändert, dass sie den um eins erhöhten Wert zurückgibt. Anschließend wird der Variable i der neue Wert zugewiesen.

```
01 #include <stdio.h>                  /*Header einbinden          */
02 int pluseins(int i);                /*Prototyp der Funktion     */
03 int main(void)                     /*Beginn, main - Funktion */
04 {
05     int i=0;                       /*dekl.und initialisieren */
06     printf("i ist: %d\n",i);       /*Testausgabe - Bildschirm*/
07     i = pluseins(i);               /*Funktionsaufruf           */
08     printf("i ist: %d\n",i);       /*Testausgabe - Bildschirm*/
09     fflush(stdin);                 /*Tastaturpuffer löschen  */
10     getchar();                     /*auf Tastatur warten       */
11     return 0;                      /*0 zurückgeben an Konsole*/
12 }
13
```

```
14 int pluseins(int i)                  /*Definition der Funktion */
15 {
16     i++;                             /*i um eins erhöhen       */
17     return i;                        /*i zurückgeben           */
18 }
```



Diese Ausgabe sieht schon besser aus, das Programm tut, was es soll! In Zeile 9 wird die i – Variable der Funktion übergeben. Dann „kopiert“ sich die Funktion die Variable und erhöht sie um 1. Bei `return i;` gibt die Funktion den Wert 1 zurück und in der `main` – Funktion wird der Variable i der zurückgegebene Wert zugewiesen. In Bild 4.2 ist der Programmablauf noch einmal dargestellt.



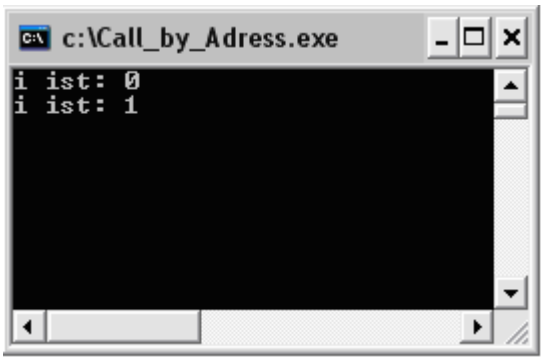
Das Programm funktioniert zwar, aber die Lösung ist nicht so elegant. Denn man könnte in Zeile 9 anstelle des Funktionsaufrufes gleich i++ schreiben und die

Ausgabe auf den Bildschirm würde genauso aussehen. Mit anderen Worten, die pluseins – Funktion ist in diesem Beispiel überflüssig.

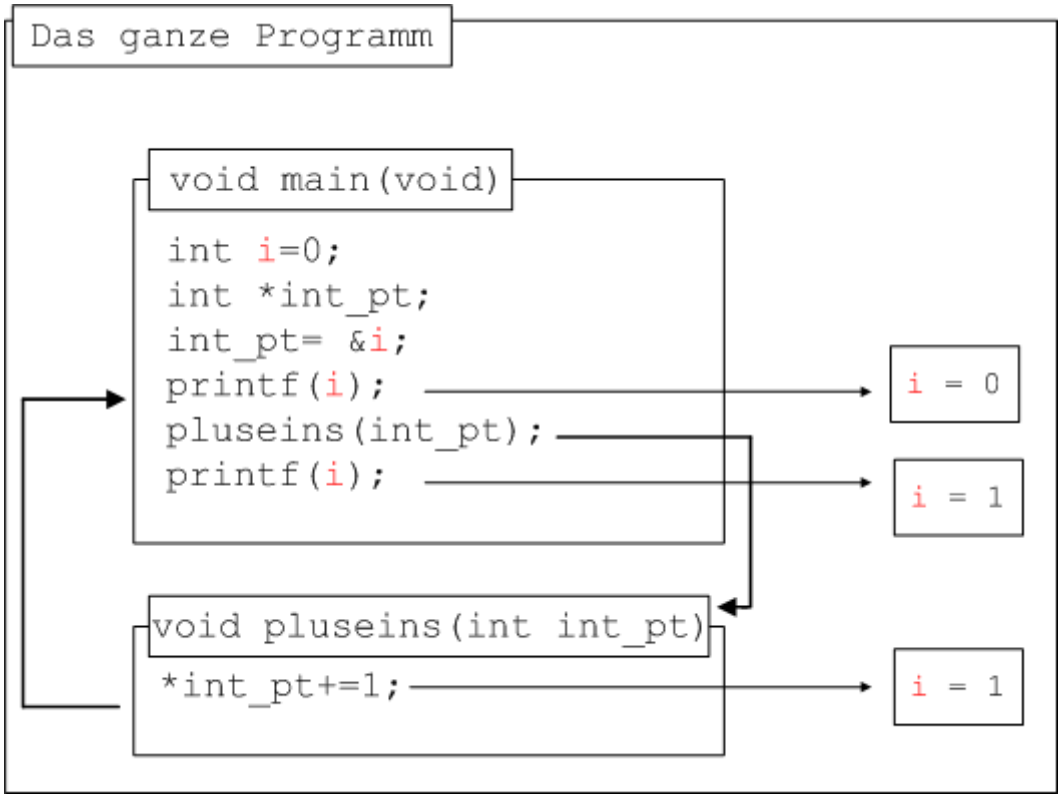
Nehmen wir aber einmal an, die Variable i soll wie im Quelltext 4.1 direkt in der pluseins - Funktion geändert werden und es soll nicht mit einer Kopie einer Variablen gearbeitet werden. Das funktioniert nur mit Pointern!

In Quelltext 4.3 ist die beschriebene Vorgehensweise beachtet worden. Es werden zunächst eine int – Variable und eine Pointer – Variable auf den Datentypen int deklariert. Dann wird dem Pointer int_pt die Adresse von i zugewiesen. Mit printf wird nun überprüft, ob der Pointer wirklich auf die Variable i zeigt. Im folgenden Schritt wird die pluseins – Funktion aufgerufen und es wird eine Adresse im Arbeitsspeicher an die Funktion übergeben. Diese „schaut“ sich die Adresse an und „kopiert“ sie sich in eine eigene Pointervariable, welche nur in der pluseins – Funktion gültig ist. Da nun in der lokalen Pointervariable eine Speicheradresse im Arbeitsspeicher steht, kann diese Speicherstelle (welche der Variable i aus der main – Funktion entspricht) von der Funktion aus manipuliert werden. Der Wert wird ausgelesen, um 1 erhöht und wieder zurück geschrieben. Um zu prüfen, ob auch alles geklappt hat, wird noch einmal eine Ausgabe auf den Bildschirm vorgenommen. Diese Art des Funktionsaufrufes wird Call-by-Adress genannt. Das Beispiel ist in Quelltext 4.3 zu sehen.

```
01 #include <stdio.h>                          /*Header einbinden */
02
03 void pluseins(int *int_pt);                /*Prototyp         */
04 int main(void)                             /*Beginn, main - Funktion */
05 {
06     int i=0;                               /*deklarieren, initialisieren */
07     int *int_pt;
08     int_pt= &i;
09     printf("i ist: %d\n",*int_pt);         /*Testausgabe      */
10     pluseins(int_pt);                      /*Funktionsaufruf  */
11     printf("i ist: %d\n",i);               /*Testausgabe      */
12     fflush(stdin);                         /*Tast.puf.löschen */
13     getchar();                             /*auf Tast. Warten */
14     return 0;                              /*0 zurückgeben    */
15 }
16
17 void pluseins(int *int_pt )                /*Definition der Funktion  */
18 {
19     *int_pt += 1;                          /* *int_pt um 1 erhöhen    */
20 }
```



In Bild 4.3 ist der Programmablauf noch einmal grafisch dargestellt.



4.2 Funktionen mit mehreren Rückgabewerten

Wenn man sich einmal einen Funktionsprototypen anschaut kann man erkennen, dass zwar mehrere Variablen an eine Funktion übergeben werden können, aber es nur möglich ist einen Wert zurückzugeben. Als Beispiel ist nun der Prototyp einer Funktion gezeigt, die zwei int – Zahlen voneinander abzieht und das Ergebnis als Rückgabewert zurückliefert.

```
int minus(int gross, int klein);
Rückgabewert Funktionsname(erste Variable, zweite Variable);
```

In diesem Fall ist es kein Problem, dass man nur einen Wert zurückgeben kann. Aber es gibt auch Fälle in denen man zwei Werte zurückgeben will, weil man z.B. den Widerstand und die Leistung von einem übergebenen Strom und einer übergebenen Spannung in einer Funktion errechnet hat. Dies funktioniert nur über Pointer. Für dieses beschriebene Beispiel würde ein Funktionsprototyp wie folgt aussehen:

```
void berechnung(double u, double i, double *p_pt, double *r_pt);
```

Man übergibt der berechnung – Funktion die Spannung und den Strom wie bei Call – by – Value und die Speicheradressen für die Leistung und den Widerstand wie bei Call – by – Adress. Dabei ist es wichtig, dass die beiden Pointer auf gültige Speicherbereiche zeigen. In der Funktion werden der Widerstand und die Leistung errechnet und über Pointer in die Variablen r_err und p_err hineingeschrieben. Im Quelltext 4.4 ist diese Art von Funktionsaufruf schrittweise dargestellt.

```
01 # include <stdio.h>
02
03 void berechnung(double u, double i, double *p_pt, double *r_pt);
04 int main (void)
05 {
06     double u_mess = 0, i_mess = 0;
07     double p_err = 0, r_err = 0;
08     double *p_err_pt = NULL;
09     double *r_err_pt = NULL;
10
11     /*Pointer setzen*/
12     p_err_pt = &p_err;
13     r_err_pt = &r_err;
14
15     /* vom Messgerät gemessene Spannung und gemessener Strom */
16     u_mess = 12.5;
17     i_mess = 1.2;
18
19     /*Leistung und Widerstand in der Funktion errechnen*/
20     berechnung(u_mess, i_mess, p_err_pt, r_err_pt);
21
22     /*Ausgabe*/
23     printf("Gemessene Werte:\n");
24     printf("Spannung: %.2f Strom: %.2f\n\n", u_mess, i_mess);
25     if(i_mess != 0)
26     {
27         printf("Errechnete Werte:\n");
28         printf("Widerstand: %.2f Leistung: %.2f\n", r_err, p_err);
29     }
30     else printf("Division durch 0 nicht moeglich!\n");
31
32     fflush(stdin);
33     getchar();
34     return 0;
35 }
```

```
36
37 void berechnung(double u, double i, double *p_pt, double *r_pt)
38 {
39     /*Widerstand ausrechnen, wenn strom != null ist und Speichern*/
40     if (i!=0)
41     {
42         *r_pt = u/i;
43     }
44
45     /*Leistung errechnen und Speichern*/
46     *p_pt = u * i;
47 }
```

Hat man mehrere Variablen eines gleichen Datentyps, werden diese oft zu Feldern oder Arrays zusammengefügt. Im Abschnitt „Pointer & Felder“ wurde die Arbeitsweise von Pointern im Zusammenhang mit Feldern bereits ausführlich erläutert. Dieses Wissen wird jetzt benötigt, da man in Funktionen auch nur über Pointer auf Datenfelder direkt zugreifen kann. Es ist im Prinzip eine Variation von „Funktionen mit mehreren Rückgabewerten“.

Die Arbeit mit Funktionen ist denkbar einfach: man übergibt der Funktion die erste Speicheradresse des Datenfeldes und eventuell die Anzahl der Feldelemente. In der Funktion muss jetzt nur noch mittels Pointerarithmetik durch das Feld „gewandert“ werden und die Werte des Datenfeldes können von der Funktion aus direkt verändert werden. Da sich dieser Satz zwar leicht liest, die Vorstellung darüber aber etwas schwierig ist, ist in folgendem Quelltext diese Funktionsweise einmal Schritt für Schritt aufgelistet. Die `feldbeispiel` – Funktion erhöht den Inhalt der Feldelemente um eins und gibt sie dann zurück.

```
01 #include <stdio.h>
02
03 void feldbeispiel(int *anfangsadresse, int anzahlfeldelemente);
04 int main(void)
05 {
06     int feld[5]={1,2,3,4,5};
07     int *feld_anfang;
08
09     feld_anfang=feld; /*oder: feldanfang = &feld[0];*/
10     feldbeispiel(feld_anfang, 5);
11 }
12
13 void feldbeispiel(int *anfangsadresse, int anzahlfeldelemente)
14 {
15     int i=0;
16
17     for(i=0;i<anzahlfeldelemente;i++)
18     {
19         *(anfangsadresse+i) = *(anfangsadresse+i)+1;
20     }
21 }
```

4.3 Dynamische Speicherplatzverwaltung, Pointer & Strukturen

Dynamische Speicherplatzverwaltung gehört zwar nicht direkt in diese Lerneinheit und auch eigentlich nicht unbedingt an diese Stelle. Da sie aber ein wesentlicher Bestandteil von Softwaretechnik im 2. Semester ist und dort überwiegend mit Komplexen Datentypen hantiert wird, ist hier ein Beispiel gezeigt, wie man über Pointer auf eine Komplexe Variable (Struktur) zugreift. Außerdem kommen wir am Ende dieses Abschnittes wieder bei Funktionen an. Ein Komplexer Datentyp besteht aus einem Verbund unterschiedlicher Datentypen. Z.B. hat der Datentyp `reifen` einen `preis` und einen `hersteller`.

```
01      typedef struct
02      {
03         short preis;
04         char hersteller[40];
05      }reifen;
```

Bei dieser Schreibweise heißt der Datentyp in der `main` – Funktion einfach `reifen`. (Bsp: `reifen fahrradreifen;`) Es gibt allerdings noch eine zweite Schreibweise.

```
01      struct reifen
02      {
03         short preis;
04         char hersteller[40];
05      };
```

In diesem Fall heißt der Datentyp `struct reifen`. (Bsp. `struct reifen fahrradreifen;`) In dieser Lerneinheit wird die erste Schreibweise benutzt. Da das Thema an dieser Stelle nur kurz angeschnitten wird, ist im folgenden Quelltext 4.6 gezeigt wie man einen komplexen Datentypen erzeugt und mit Inhalt füllt. Dies wird einmal mit Pointer und einmal ohne Pointer gemacht. Es ist zu beachten, dass es zwei Schreibweisen gibt, um mit Pointern und Strukturen zu arbeiten. Zum Einen der Punktoperator (zuerst im Beispiel) und den Pfeiloperator (zweite Variante im Beispiel). Der Einfachheit halber ist zu empfehlen, den Pfeiloperator zu verwenden.

```
01 #include<stdio.h>
02 #include<string.h>
03
04 typedef struct
05 {
06     short preis;
07     char hersteller[40];
08 }reifen;
09
10 int main(void)
11 {
```

```
12      /*Variablen deklarieren*/
13      reifen fahrradreifen = {0, "Hersteller" };
14      reifen autoreifen = {0,"Hersteller" };
15      reifen *pt_f, *pt_a;
16
17      /*Pointer setzen*/
18      pt_f=&fahrradreifen;
19      pt_a=&autoreifen;
20
21      /*Werte setzen ohne Pointer (Strings über strcpy)*/
22      fahrradreifen.preis = 10;
23      strcpy(fahrradreifen.hersteller, "Continental");
24      autoreifen.preis = 100;
25      strcpy(autoreifen.hersteller, "GoodYear");
26
27      /*Werte setzen mit Pointer*/
28      (*pt_f).preis = 9;
29      strcpy((*pt_f).hersteller, "Maxxis");
30      (*pt_a).preis = 99;
31      strcpy((*pt_a).hersteller, "Michelin");
32
33      /*Werte über Pointer setzen 2*/
34      pt_f->preis = 8;
35      strcpy(pt_f->hersteller, "Vittoria");
36      pt_a->preis = 88;
37      strcpy(pt_a->hersteller, "Blackhawk");
38
39      fflush(stdin);
40      getchar();
41      return 0;
42  }
```

Felder von Strukturen lassen sich ganz einfach wie gewöhnliche Felder „durchwandern“. Hätte man z.B. ein Feld von 10 fahrradreifen – Variablen, könnte man mit einer Zählervariable durch das Feld „wandern“ und die Inhalte wie im folgenden Quelltext 4.7 gezeigt, bearbeiten. Dieser Quelltext zeigt die Variante, bei der der Zeiger „verbogen“ wird. Es ist aber auch die Variante möglich, in der der Pointer direkt durch das Feld „wandert“.

```
01 #include<stdio.h>
02 #include<string.h>
03
04 typedef struct
05 {
06      short preis;
07      char hersteller[40];
08 }reifen;
09
10 int main(void)
11 {
12      /*Variablen deklarieren*/
13      int i=0;
14      reifen fahrradreifen[10];
15      reifen *pt_f;
16
17      /*Pointer setzen*/
```

```
18      pt_f=&fahrradreifen[0];
19
20      for(i=0;i<10;i++)
21      {
22        (*(pt_f+i)).preis = 9;
23        strcpy((*(pt_f+i)).hersteller, "Maxxis");
24      }
25
26      for(i=0;i<10;i++)
27      {
28        (pt_f+i)->preis = 8;
29        strcpy((pt_f+i)->hersteller, "Vittoria");
30      }
31
32      fflush(stdin);
33      getchar();
34      return 0;
35  }
```

Da nun der Umgang mit Strukturen und Pointern gezeigt wurde kann man sagen, dass der Umgang in Funktionen mit Strukturen genau so funktioniert, wie mit gewöhnlichen Variablen oder Feldern. Es wird Ausschließlich eine (Anfangs-) Adresse an eine Funktion übergeben und diese speichert die Daten, die in der Funktion erzeugt werden direkt an diese Speicherstelle (oder „wandert“ gegebenen Falls durch das Feld). Der Quelltext 4.8 zeigt dies noch einmal. Es wird ein Feld von 10 fahrradreifen deklariert. In der Funktion fuellen wird das Feld mit leeren Werten gefüllt.

```
01 #include<stdio.h>
02 #include<string.h>
03
04 typedef struct
05 {
06      short preis;
07      char hersteller[40];
08 }reifen;
09
10 void fuellen(reifen *pt, int anzahl);
11
12 int main(void)
13 {
14      /*Variablen deklarieren*/
15      int anzahl=10;
16      reifen fahrradreifen[10];
17      reifen *pt;
18
19      /*Pointer setzen*/
20      pt=&fahrradreifen[0];
21
22      fuellen(pt, anzahl);
23
24      fflush(stdin);
25      getchar();
26      return 0;
27 }
```

```
28
29 void fuellen(reifen *pt, int anzahl)
30 {
31     int i;
32     for(i=0;i<anzahl;i++)
33     {
34       (pt+i)->preis = 0;
35       strcpy((pt+i)->hersteller, "default");
36     }
37 }
```

4.4 Fragen

1. Die `main()`-Funktion hat immer einen Rückgabewert. (falsch)
2. Bei Variablen und Funktionen muss man auf den Gültigkeitsbereich achten. (wahr)
3. `void funktion(int *pt);` → Funktionsaufruf mit Call-by-Adress. (wahr)
4. Bei Funktionen sind Standardmäßig mehrere Rückgabewerte vorgesehen. (falsch)
5. Bei Call-by-Value „kopiert“ die Funktion sich die Variablen. (wahr)
6. Es können nur Pointer oder nur Variablen an eine Funktion übergeben werden. (falsch)
7. Beim bearbeiten von Feldern wird die Anfangsadresse an die Funktion übergeben. (wahr)
8. Strukturen bestehen aus dem Verbund mehrerer Variablen des gleichen Typs. (falsch)
9. Auch ein Pointer auf eine Struktur kann an eine Funktion übergeben werden. (falsch)
10. In Funktionen können höchstens 9 Pointer übergeben werden. (falsch)

4.5 Aufgaben

4.5.1 Zahlentausch

Schreibe ein Programm, in dem 2 `float` Zahlen von der Tastatur eingelesen werden. Diese sollen in einer `tausche` - Funktion miteinander vertauscht werden. Gib die beiden Variablen vor und nach dem Tauschen zur Überprüfung auf dem Bildschirm aus.

4.5.2 Kaugummiautomat

Schreibe ein Programm für einen Kaugummiautomaten. Folgende Funktionsprototypen sind gegeben:

```
void aufgeldwarten(int *geldeinwurf_pt);
void kaugummiausgeben(int *geldeinwurf_pt, int *geldstand_pt, int
*kaugummistand_pt);
```

Ein „Sensor“ (Eingabeaufforderung über `if` - Abfrage) soll überprüfen, ob 1 € eingeworfen worden ist. Wurde der Automat mit 1 € gefüttert, so soll

1. *Ein Kaugummi ausgegeben werden (natürlich Symbolisch)*
2. *Der Zählerstand für die Anzahl der Kaugummis um eins verringert werden*
3. *Der Zählerstand für sie Summe des Geldes um 1 € erhöht werden*
4. *Das Programm in den „Sensor“ Modus zurückkehren*

Wenn keine Kaugummis mehr im Automaten sind oder die Geldkassette des Automaten mit 100 € gefüllt ist, soll ein Warnhinweis erscheinen mit der Bitte Kaugummis aufzufüllen oder die Geldkassette zu leeren.

4.5.3 Vektor 3D

Schreibe ein Programm, in dem 2 dreidimensionale Punkte im Raum definiert sind. Das Programm soll eine Funktion enthalten, die den Vektor zwischen den Punkten bestimmt und den Betrag des Vektors ausrechnet. Der Betrag des Vektors soll als Rückgabewert der Funktion zurückgegeben werden und der Vektor soll über Pointer so ausgegeben werden, dass er auch in der `main` - Funktion abrufbar ist. Nach dem Funktionsaufruf sollen die beiden Punkte und der Vektor, sowie der Betrag des Vektors auf den Bildschirm ausgegeben werden. Für die Berechnungen in der Funktion werden die Befehle "`double pow(double x, double y);`" und "`double sqrt(double x);`" benötigt, die in der Header - Datei maht.h stehen. Als Hilfestellung ist folgender Funktionsprototyp gegeben:

```
double abstand_3d_zwischen_a_und_b(int *a, int *b, int *c);
```

5. Quellenverzeichnis

[1] http://www.galileocomputing.de/openbook/c_von_a_bis_z/

[2] Rolf Isernhagen, Hartmut Helmke; Softwaretechnik in C und C++ - Das Kompendium; Auflage 3; Hanser Fachbuchverlag, 2004

[3] RRZN; Die Programmiersprache C. Ein Nachschlagewerk; Auflage 14

[4] E. Forgber; Einführug in die Programmiersprache C; Vorlesungsskript FH Hannover; 2005