Weiteres zu Zeigern

Nice to Know

Beispiel für das Verwenden nicht typisierter Zeiger

```
void copy(void* ziel, const void * quelle, int count) {
   char* y = reinterpret_cast<char*>(ziel);
   char* x = reinterpret_cast<char*>(quelle);
   for (i=0; i < count; ++i) {
      *y = *x; ++y; ++x; }
}
. . .
Somewhat a, b;
int i, j;
. . .
copy(&b, &a, sizeof(Somewhat));
copy(&j, &i, sizeof(int));                    // entspricht j = i
```

Regeln zur Konvertierung von Zeigern

in C++ ist ein explizites type cast von **(void*)** in **(*type**)** erforderlich.
In C++ sollte dafür der Operator ***reinterpret_cast*** verwendet werden.

void*-Zeiger (d.h. nichttypisierte Zeiger) sind mit allen typisierten Zeigern kompatibel, explizite *type casts* der Form **(void*)** und **(*type**)** sind in also C optional;

Bei Konvertierungen zwischen typisierten Zeigern ist die explizite Angabe des *type casts* der Form **(*type**)** bzw. ***reinterpret_cast*** zwingend vorgeschrieben;

Zeiger auf Funktionen (Funktionszeiger) können **nicht** konvertiert werden.

überspringen

Noch ein paar Nice-to-Knows

reinterpret_cast-Operator

Der ***reinterpret_cast***-Operator erlaubt es, einen Zeiger auf einen beliebigen Typ in einen Zeiger auf einen beliebigen anderen Typ zu konvertieren. Der C-Allzweck-Konvertierungsoperator (*Zieltyp*) *Ausdruck* erlaubt das natürlich auch, ist aber nicht so aussagekräftig:

```
typedef struct { char b1, b2, b3, b4; } BYTES;
int  ix = 'a' + 256 * 'b' + 256*256 * 'c' + 256*256*256 * 'd';

BYTES* pbx = reinterpret_cast<BYTES*>(&ix);
// BYTES* pbx = (BYTES*)(&ix);                    // C-Allzweck-Cast

cout << pbx->b1 << pbx->b2 << pbx->b3 << pbx->b4;
// Es werden die vier Bytes von ix als char interpretiert und ausgegeben
```

reinterpret_cast kann nur zur Konvertierung von Zeigern verwendet werden. Zur Konvertierung zwischen elementaren Datentypen wird ***static_cast*** verwendet.

Programm zur Ausgabe der Binärdarstellung von int

```
   int j = 1026;
   int* pint = &j;
   char* pch = reinterpret_cast<char*>(pint);
   for (int i=0; i < 4; ++i, ++pch)   {
      cout << " Byte " << i << ":" << static_cast<int>(*pch) << endl;
   } // for i

// Ausgabe ist:  Byte 0: 2  Byte 1: 4  Byte 2: 0  Byte 3: 0
```

Übung (evtl. weglassen)

Ermitteln Sie die Binärdarstellung einer float-Variablen im Speicher.

Welche Binärdarstellung entspricht z.B.

- 0.5;
- 2,
- 4.0 sowie
- -4.0 sowie
- 0.875 sowie
- 7

Wo steckt z.B. das Vorzeichen der Zahl und wo das des Exponenten?

Zeiger - Speicherbedarf von Zeigern

Der Speicherplatz, der für einen Zeiger reserviert wird, muss eine Speicheradresse aufnehmen können.
Das bedeutet, dass ein Zeiger des Typs *int* und ein Zeiger auf einen Datentyp *double* normalerweise gleich groß sind.
Der Typ, der einem Zeiger zugeordnet ist, gibt den Inhalt und damit auch die Größe des adressierten Speicherbereichs an.

```
int* pint ;              // pint belegt 4 Byte          (*)
double* pdouble;         // pdouble belegt auch 4 Byte  (*)
```

(*) Der für Zeiger vorgesehene Speicherplatz ist natürlich implementierungsabhängig. Der Wert von 4 Byte ist der für heutige 32-Bit-Systeme in der Regel verwendete.

Zeiger auf Funktionen und komplexe Deklarationen

Zeiger auf Funktionen

Deklaration eines Zeigers auf eine Funktion ohne Parameter bzw. eines entsprechenden Typs:

```
void (*func1) (void);
typedef void  (*Tfunc1) (void);
```

Deklaration eines Zeigers auf eine Funktion mit zwei Parametern bzw. eines entsprechenden Typs:

```
int (*func2) (char*, int);
typedef int  (*Tfunc2) (char*, int);
```

```
typedef double Meter;

Meter  laenge;

typedef Vorgang*   T_intZeiger


void Mit3040_Belegen(Vorgang*&  pz1,    Vorgang*&  pz2);
void Mit3040_Belegen(T_intZeiger&  pz1, T_intZeiger&  pz2);



typedef map<int, map<vector<int>, set<int,greater<int>>>
        T_StudentenGrab;
```

Zeiger auf Funktionen (2)

```
#include <iostream>
using namespace std;

typedef double (*func) (double);

double f2(double x) { return x*x; }
double f3(double x) { return x*x*x; }
double f4(double x) { return x*x*x*x; }
double f5(double x) { return x*x*x*x*x; }

func liste[4] = { f2, f3, f4, f5 };  /* { &f2, &f3, &f4, &f5 }; geht auch */

int main(void) {
   int i;
   for (i=0; i<4; ++i) {
      double y = liste[i](5); cout << y << endl; }  /* *liste[i](5); */
   return 0;
}
```

Zeiger auf Funktionen (3)

Eine fortgeschrittene und in vielen Fällen nützliche Programmiertechnik ist die Übergabe von Zeigern auf Funktionen als Funktionsparameter. Man schreibt z.B. eine Funktion zur Berechnung der Nullstelle einer anderen Funktion, die man dann als Parameter übergibt, z.B:

```
/*
  double sin(double);  in math.h
*/
float Nullstelle (double x1, double x2, double (*f) (double x));
. . .
x0 = Nullstelle(6, 7, sin);  /* x0 == 2* pi */
```

Komplexe Deklarationen

double *x[100];

ist x kein Zeiger auf einen Vektor mit 100 Elementen vom Typ double, sondern **ein Vektor mit 100 Elementen vom Typ Zeiger auf *double***!
Grund: Der Operator ***[]*** hat eine höhere Priorität hat als der Operator * !

double (*x)[100];

dagegen ist x ein **Zeiger auf einen Vektor mit 100 Elementen vom Typ *double***!

int (*f)();

spezifiziert einen Zeiger auf eine Funktion f, der z.B. als Feld in einer Struktur oder als Funktionsparameter verwendet werden kann

char* (*f[10])(int, int (*f1)(long, long));

und was ist das?

Algorithmus zur Analyse komplexer Deklarationen

Was ist also das?

```
char* (*f[10])(int, int (*f1)(long, long));
```

1. Beginne beim Variabelennamen, also bei ***f***.
2. Es werden nun von **links nach rechts** runde und eckige Klammernpaare interpretiert.
3. Wird das rechte Ende erreicht, werden die noch nicht interpretierten Zeichen von **rechts nach links** interpretiert.
4. Beim Erreichen einer rechten schließenden runden Klammer, wird die Interpretationsrichtung umgekehrt.
5. Bei Erreichen einer linken öffnenden Klammer, wird wieder bei Schritt 2 fortgesetzt.

Algorithmus zur Analyse komplexer Deklarationen (2)

int (*f1)(long, long);

f1 ist ein Zeiger auf eine Funktion mit zwei Parametern vom Typ *long* und dem Rückgabewert vom Typ *int*.

char* (*f[10]) (...);

f ist ein Feld mit 10 Elementen vom Typ Zeiger auf Funktion mit den Parametern ... und dem Rückgabetyp Zeiger auf *char*.

char* (*f[10])(int, int (*f1)(long, long));

f ist ein Feld mit 10 Elementen vom Typ Zeiger auf Funktion mit den Parametern vom Typ *int* und vom Typ Zeiger auf Funktion mit zwei Parametern vom Typ *long* und dem Rückgabewert int. Der Rückgabewert der Funktion ***f*** ist vom Typ Zeiger auf *char*.

Rechnen mit Zeigern

Addition und Subtraktion bei Zeigern

```
int i;                  // z.B. Adresse 0x0100
int j;                  // z.B. Adresse 0x0104
int k;                  // z.B. Adresse 0x0108
int arr[10];            // z.B. Adresse 0x010C, 0x0110, 0x0114 ...

int* ptr = & i;         // ptr enthält Adresse von i, d.h 0x0100

ptr = ptr + 2;          // ptr enthält nun 0x0108, d.h. Adresse von k
++ptr;                  // ptr enthält nun 0x010C, d.h. Adresse arr[0]
```

Bei der Addition (Subtraktion) eines ganzzahligen Wertes (z.B. 2), wird der Wert der entsprechenden Adresse um die Größe von z.B. zwei Objekten dieses Datentyps (hier int) erhöht (erniedrigt). Wenn z.B. ein char 1 Byte, ein int 4 Byte und ein double 8 Byte belegt, dann erhöht sich bei einer Addition um 2 zu einem Zeiger, der Wert der im Zeiger gespeicherten Adresse um 2, 8 bzw. 16 Byte.

Operationen mit Zeigern

Einem Zeiger kann man einen Zeiger des gleichen Typs zuweisen.
Ein Zeiger kann auch inkrementiert oder dekrementiert werden.
So bedeutet z.B. „+5“ hier eine Inkrementierung um 5 Schritte mit der Schrittweite *„sizeof(int)“:*

```
int* ptr1 = &i;
int* prt2 = ptr1;
int* ptr3 = ptr2 + 5;
```

Eine Zuweisung eines anderen Typs ist nur über Typecast möglich:

```
char* pchar = (char*) ptr2;  // typecast

char* pchar = reinterpret_cast<char*>(ptr2);  // in C++ empfohlen
```

Operationen mit Zeigern (2) Differenzbildung

Es kann die Differenz zweier Zeiger gebildet werden.

```
int arr[10];
int* ptr1 = &arr[1];
int* ptr8 = &arr[8];
int anzahl = ptr8 – ptr1;  // ergibt den Wert 7
// Besser:
ptrdiff_t anzahl = ptr8 – ptr1;
// ptrdiff_t ist in cstddef (bzw. stddef.h) vereinbart
```

Einem Zeiger darf der Wert *0 (NULL)* zugewiesen werden. Dadurch wird der Zeiger als nicht initialisierter Zeiger gekennzeichnet:

```
int* ptr = 0;  // oder: int* ptr = NULL;
```

Die Vergleichsoperationen == und != sind zulässig, um zwei Zeiger auf Gleichheit zu testen, d.h. ob sie auf die gleiche Adresse verweisen. Dieses hat nichts mit den Werten, die die Adressen enthalten, zu tun.

Übung: Was macht das folgende Programm?

```
#include <iostream>
using namespace std;
int main()
{
int i = 7;
int j = 16;
int* pint = & i;
*pint = 22;

cout<< "pint selbst ist " <<pint
    << "*pint ist " <<*pint
     << "    i ist "  << i
     << "    j ist " << j;
pint = &j;
cout << "pint selbst ist " <<pint
    << "*pint ist " <<*pint
     << "    i ist "  << i
     << "    j ist " << j ;
```

```
int** ppint = & pint;

*pint = 18;
cout << "**ppint ist "
   << **ppint
   << "*pint ist "  <<*pint
   << "    j ist "  << j;

j = 66;
cout << "**ppint ist "
     << **ppint
     << "*pint ist "  << *pint
     << "    j ist "  << j;
return 0
}
```

Übung zu Zeiger und Funktionen

Übung

Erklären Sie, was das folgende Programm macht?

```
int main(){
        int N=5;
        int arr[100] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        cout << "&arr[0] " << &arr[0] << endl;
        cout << "&arr[9] " << &arr[9] << endl;
        cout << "&N " << &N << endl;

        arr[-1] = 95;
        for (int i=1; i < N; ++i) {
                cout << arr[i] << " ";
        }

}
```

Als Release compiliert.

Zeichenketten

Zeichenketten

Arrays können von einem beliebigen Typ (auch einem selbstdefinierten) angelegt werden:

```
int  GanzzahligesFeld[100];
short  KleinePrimZahlen[8];
bool Entscheidungen[4]={true, false, true, true};

char MeinName[]= "Hartmut Helmke";
```

Zeichenketten, d.h. Arrays des Typs “char” werden auch Strings genannt. Sie werden mit dem Zeichen 0 ('\0‘) abgeschlossen.
Die leere Zeichenkette besteht also mindestens aus einem Zeichen.
Sie können durch strcpy, strlen etc. manipuliert werden

Funktionen zur Zeichenkettenverarbeitung

include von <string.h> bzw. <cstring> erforderlich

```
char*  strcpy(char* ziel,  const char* quelle);

size_t strlen(const char* string);

char*  strcat(char* ziel,  const char*  str2);

int strcmp(const char*  str1,  const char* str2);

int strncmp(const char*  str1,  const char* str2, const size_t len);
```

size_t ist ein ganzzahliger vorzeichenloser und implementierungs-abhängiger Typ, der zu den ganzzahligen Standardtypen kompatibel ist.

überspringen

Initialisierung von Zeichenketten

Nur bei der Definition von Zeichenketten ist eine Wertzuweisung in doppelten Anführungsstrichen zulässig

```
char meinName[] = “Hartmut Helmke“;   // 15 Byte groß (inklusive '\0‘)
```

Im Programm ist eine Zuweisung an eine Zeichenkette nur durch strcpy und entsprechende Funktionen möglich.

```
meinName = “Meier“;                 // Syntaxfehler

strcpy(meinName, “Meier“);          // richtig

meinName[0]=‘M‘;                     // ebenfalls
meinName[1]=‘e‘;                     // richtig
. . .
```


überspringen

Zeiger und typedef

typedef dient zur Deklaration eines neuen Typnamens, zum Beispiel:

```
typedef double  T_Meter;
typedef enum {false=0,true=1} bool;
typedef T_Meter  MeineEinheiten;
```

Der erste Teil ist der bekannte Teil, der zweite der neue Name.

Die Deklaration und Verwendung eines Zeigertyps sieht wie folgt aus:

```
typedef int*  TP_int;       // Zeiger auf int-Variablen
TP_int  p_int;              // entspricht int* p_int
```

Entsprechend Deklaration und Verwendung eines Vektortyps:

```
typedef char  Name[20];
Name meiner;                // entspricht char  meiner[20];
```


überspringen

Übung zum Thema *Referenzierung/Dereferenzierung*

```
void swap(float* x1, float* x2) {              // zwei Werte vertauschen
   float temp = *x1; *x1 = *x2; *x2 = temp;
}

void swap2(float** x1, float** x2) {           // zwei Zeiger vertauschen
   float* temp = *x1; *x1 = *x2; *x2 = temp;
}

int main(void) {
   float a, b, *pa, *pb;
   a = 1.0, b = 2.0, pa = &a, pb = &b;
   /*(1)*/ swap(&a, &b);            /* welche dieser          */
   /*(2)*/ swap2(&a, &b);           /* Aufrufe sind           */
   /*(3)*/ swap2(&pa, &pb);         /* syntaktisch richtig,   */
   /*(4)*/ swap2(&&a. &&b);         /* welche sind            */
   /*(5)*/ swap2(&(&a), &(&b));     /* syntaktisch falsch ?*/
   . . .
}
```

Lösung

(1) ist richtig

(2) ist falsch in C++ und nicht falsch in C (kein Fehler, nur Warnung), aber semantisch unklar

(3) ist richtig

(4) ist falsch:
(a) die Zeichenfolge && ist uneindeutig (&&: Log. UND)
(b) &a sowie &b sind keine Lwerte (Lvalues)

(5) ist auch falsch entsprechend 4(b) !

L-Wert (L-Value: Location for a Value, *LB S. 24, KP S. 30*)

- Ein **L-Wert** ist ein **Objekt**, z.B. eine Variable, dem ein Bereich im **Arbeitsspeicher** zugeordnet ist, dem ein **Wert** zugewiesen werden kann !

```
int a=1, b=1, *p = ...;

a = 27;          /* i.O. */
a + 1 = 27;      /* linker Operand ist kein L-Wert */
++b = 27;        /* linker Operand ist kein L-Wert */
p++;             /* i.O. */
*p++ = 27;       /* i.O. ( *p=27; p++; )    */
*++p = 27;       /* i.O. ( p++; *p=27; )    */
(p++)++;         /* p++ ist kein L-Wert   */
p = &a;          /* i.O. */
&a = &b;         /* linker Operand ist kein L-Wert */
```

Mehrdimensionale Arrays

Mehrdimensionale Vektoren

Es lassen sich auch mehrdimensionale Vektoren definieren. Jede Dimension muss in separaten eckigen Klammern angegeben werden:

```
float Kontobew[3][4];
```

Die erste Dimension ist die *Zeilenanzahl*. Es wird ein zweidimensionaler Vektor mit drei Zeilen mit jeweils vier Spalten deklariert.

Bei der Speicherbelegung „*läuft*“ die hinterste Dimension zuerst.

Initialisierung von mehrdimensionalen Arrays

```
float Kontobew[3][4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

Die inneren geschweiften Klammern sind redundant, erleichtern jedoch die Lesbarkeit. Die folgende Initialisierung wäre daher gleichwertig:

```
float Kontobew[3][4] = { 1,2,3,4,5,6,7,8,9,10,11,12 };
```

Ein mehrdimensionales Array wird also in der Form a00, a01, a02... a10... im Speicher abgelegt. Durch folgende Deklaration wird das erste Element jeder Zeile initialisiert. Die restlichen Elemente erhalten den Wert 0.

```
float Kontobew[3][4] = { {1 }, {5 }, {9 } };
```

```
float Kontobew  []  [4] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

Das ist richtig, das Folgende ist allerdings ein Fehler:

```
float Kontobew[3] [] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12} };
```

Zugriff auf die Elemente von mehrdimensionalen Arrays

Der Zugriff auf die Elemente erfolgt durch *Kontobew **[i] [j].***
*Kontobew**[1, 2]*** ist zwar syntaktisch auch erlaubt, aber ergibt wahrscheinlich etwas völlig anderes als das Erwartete. Das Komma wird hier als Kommaoperator angesehen. Der Kommaoperator gibt das Ergebnis der letzten Operation als sein Ergebnis zurück. Hier also Kontobew[2], d.h. die dritte Zeile des Arrays.

Übung zu mehrdimensionalen Arrays:
Matrixmultiplikation

Beispiel: zweidimensionale Vektoren als Parameter

```
enum { ROWS=5,   /* 5 Zeilen  */
       COLS=2  /* 2 Spalten */ };

int vector[ROWS][COLS] = { {0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9} };

int f1 (int cols, int* array, int i, int j)  { return array[i*cols + j]; }
int f2 (int (*array)[COLS], int i, int j) { return array[i][j]; }
int f3 (int array[][COLS], int i, int j)   { return array[i][j]; }


. . .
int i, j, x1, x2, x3;
. . .
x1 = f1(COLS, reinterpret_cast<int*>(vector), i, j);      /* cast erforderlich */
x2 = f2(vector, i, j);
x3 = f3(vector, i, j);
. . .
```