Arrays (Felder)

```
int arrayname[n];
```

- Ein Array dient zur Speicherung größerer Datenmengen des gleichen Datentyps
- Arrays werden mit eckigen Klammern indiziert
- Um auf ein einzelnes Element zuzugreifen, ist der Name des Arrays und den Index des Elements benötigt
- In C beginnt die Nummerierung bei Null (also maximale Länge n-1)
- Beim Anlegen wird die Speichergröße festgelegt
- Es ist auch möglich Arrays variabler Größe anzulegen
- Alle Feldelemente, die nicht ausdrücklich initialisiert wurden, bekommen automatisch den Wert 0.

Arrays – Beispiel

```
int dat[10], i;
 for (i = 0; i < 10; i++)
   dat[i] = 0;
 for (i = 0; i < 10; i++)
 {
   printf ("%d. Element: ", i);
   scanf ("%d", &dat[i]);
   if (dat[i] == 0)
   {
     printf ("\n Die Eingabe war: \n");
     break;
   }
 }
 for (i = 0; i < 10; i++)
 {
   if (dat[i] != 0)
     printf ("%d. Element: %d\n", i, dat [i]);
   else
     break;  }
```

Mehrdimensionale Arrays

```
int arrayname[n][m];      Feld mit n Zeilen und m Spalten
```

- Mehrdimensionale Arrays erhält man durch mehrere Klammern

- Arrays unflexibel was die Anzahl der Elemente angeht.
- Die Anzahl der Elemente muss vor Ablauf des Programm bekannt oder überdimensioniert sein (C90)
- variable-size arrays in C99 verfügbar
- alles auf Stack allokiert $\rightarrow$ kann schnell zu “stack overflow” führen
- Lösung dursch ’dynamische Datenstrukturen’ (`malloc, free`.)
- dynamische Datenstrukturen auf heap allokiert

Mehrdimensionale Arrays – Beispiel

```
#include <stdio.h>
int main(void) {
   int i,j;
   int name[n][m];
   for(i=0; i < n; i++) {
      for(j=0; j < m; j++) {
         printf("Wert fuer name[%d][%d]:", i, j);
         scanf("%d", &name[i][j]);
      }
   }
   printf("\nAusgabe von name[%d][%d]...\n\n", n,m);
   for(i=0; i < n; i++) {
      for(j=0; j < m; j++) {
         printf("\t%4d ",name[i][j]);
      }
      printf("\n\n");
   }
}
```

Strings – Zeichenketten

```
char string[] = "Ballon";
// char string[7] = "Ballon"; // equivalent zu voriger Zeile
string[0] = 'H';
string[5] = 0;
printf("%s\n", string); → Hallo
```

- Strings: eine Folge (Arrays) von Typ **char** (oder pointer **char**$*$)
- Jedes Element des Arrays ist für ein Zeichen zuständig
- Einzelne Zeichen werden im Arbeitsspeicher nacheinander abgespeichert
- Das String-Ende wird durch eine Null markiert (0 oder '\0') die auch auf ein Byte Speicherplatz zuweist
- Es ist einfach, mit Strings Speicherzugriffsfehler zu bekommen.

Strings – Beispiel

```
 char datain[255];
  printf ("Text eingeben:\n");
  scanf ("%s", datain);
// fgets (datain, 255, stdin);
  printf ("Text ausgeben:\n%s\n",datain);
  return 0;
```

- `scanf()` liest immer nur bis zum Auftreten des ersten Leerzeichens (bis zu einem Zeilenumbruch (\n))
- Abhilfe durch `fgets()` (in stdio.h) ohne &
- `fgets()` verlangt maximale Länge, sowie Stream von dem gelesen werden soll.

Funktionen

Rückgabetyp Funktionsname(Parameterliste)
{
Anweisungen
}
wobei Parameterliste=typ1 arg1 typ2 arg2,...

- Modularisierung: das Programm wird in mehrere Programmabschnitte (Module oder Funktionen) zerlegt
- Vorteile:
  - Fehler lassen sich leichter finden
  - Bessere Lesbarkeit
  - Wiederverwendbarkeit
- Funktionstyp: sagt Compiler dass ein Werte mit dem bestimmten Typ zurückgegeben wird

Funktionen-Beispiel

```
cpstring(s1,s2)
   char s1[], s2[];
{
  int i;
  for (i=0;(s2[i]=s1[i])!='\0';i++);
return(s1,s2);
}
```

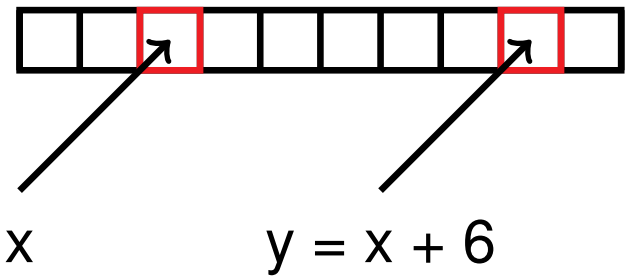
Die Funktion cpstring() kopiert ein char Feld (array).

- return: Funktion wird beendet und ein Wert an die aufrufende Funktion wird zurückgegeben
- Der Datentyp des Ausdrucks muss mit dem Typ des Rückgabewertes des Funktionskopfs übereinstimmen
- Ist der Rückgabetyp **`void`**, gibt die Funktion nichts zurück
- **`return`** verlässt eine Funktion vorzeitig (bei Rückgabetyp **`void`**)
- **`return`** `wert` liefert zusätzlich `wert` zurück

Variablen ansprechen

- Bisher: Variablen wurden immer direkt über ihren Namen angesprochen
- Intern im Rechner: Variablen werden immer über eine Addresse angesprochen (wenn Variable nicht in einem Prozessorregister befindet)
- also die Variablen bleiben im aufrufenden Code stets unverändert
- Die Speicherzellen erhalten eine Addresse
- eine Variable kann auch direkt über diese Addresse angesprochen werde
- Abhilfe: Übergabe der Speicheradresse der Variablen in Zeiger
- Bei Zeigern führt das zu Zeigern auf Zeiger (typ $\star\star$) usw.

Datentypen – Zeiger



- Zeigervariablen (Pointer) zeigen auf Speicherstellen
- Ihr Datentyp bestimmt, als was der Speicher interpretiert wird (**`void`** $*$ ist unspezifiziert)
- Es gibt keine Kontrolle, ob die Speicherstelle gültig ist (existiert, les-, oder schreibbar)
- Pointer verhalten sich wie Arrays,
  bzw. Arrays wie Pointer auf ihr erstes Element

Datentypen – Zeiger

```
int a, *b;
b=&a;
```

- Zeiger werden mit einem führendem Stern deklariert
- Bei Mehrfachdeklarationen: genau die Variablen mit führendem Stern sind Pointer
- `+,-,+=,-=, ++,--` funktionieren wie bei Integern
- `p += n` z.B. versetzt p um n Elemente
- Pointer werden immer um ganze Elemente versetzt
- Datentyp bestimmt, um wieviel sich die Speicheradresse ändert

- Der unäre Adressoperator & (Referenzoperator) bestimmt die Adresse der Variable: & Variablenname
- Der unäre Zugriffsoperator $\star$ (Dereferenzoperator) erlaubt den (indirekten) Zugriff auf den Inhalt, auf den der Pointer zeigt: $\star$ pointer
- Die Daten können wie Variablen manipuliert werden.

Zeiger – Referenzieren und Dereferenzieren

```
float *x;
float array[3] = {1, 2, 3};
x = array + 1;
printf("*x = %f\n", *x); // →*x = 2.000000
float wert = 42;
x = &wert;
printf("*x = %f\n", *x); // →*x = 42.000000
printf("*x = %f\n", *(x + 1)); // undefinierter Zugriff
```

- `*p` gibt den Wert an der Speicherstelle, auf die Pointer `p` zeigt
- `*p` ist äquivalent zu `p[0]`
- `*(p+1)` ist äquivalent zu `p[1]`
- `*(p + n)` ist äquivalent zu `p[n]`
- `&v` gibt einen Zeiger auf die Variable `v`

Beispiele – mit/ohne Zeiger

[1] Die Länge eines char Feldes ausgeben.

Ohne Zeiger:

```
length(s)
 char s[];{
   int n;
   for (n=0; s[n] !='\0';)
    n++;
   return(n);      }
```

[2] Ein char Feld kopieren.

```
cpstring(s,t)
   char *s,*t;{
     while(*t++=*s++);
      }
```

Mit Zeiger:

```
length(s)
  char *s;{
   int n;
   for (n=0; s* !='\0';s++)
   n++;
return(n);    }
```

```
cpstring(s1,s2)
   char s1[]s2[];
{
 int i;
 for (i=0;(s2[i]=s1[i])!='\0';i++);
}
```

Funktionen – Argumente

- C: ‘call by value’
- When man eine Funktion wie f(x) aufruft, der Wert von x und nicht seine Adresse wird übergeben,
- also man kann x nicht innerhalb von f ändern
- kein Problem wenn x ein array (x ist dann sowieso eine Adresse), aber wenn x eine Skalarvariable?

```
flip(x,y)
  int*x,*y;{
    int tmp;
    tmp=*x;
    *x=*y;
    *y=tmp;
    }
```

- flip wird dann aufgerufen: flip(&a,&b).

Argumente aus der Kommandozeile

`main(argc, argv[])`

- Argumente aus der Kommandozeile beim Start an Programm übergeben
- argc(**arg***ument* **c***ount*): Anzahl der Argumente auf der Kommandozeile, mit der das Programm aufgerufen wurde
- argv(**arg***ument* **v***ector*): Zeiger auf einen Vektor von Zeigern auf Zeichenketten, die die Argumente enthalten (ein Argument pro Zeichenkette)
- argv[0]: Name mit dem das Programm aufgerufen wurde → argc wenigstens 1
- argc=1, gibt keine Argumente nach dem Programmnamen auf der Kommandozeile
- Das erste optionale Argument ist argv[1] und das letzte argv[argc-1]

z.B.: `echo hello world`:

argc=3, argv[0]=`echo`, argv[1]=`hello`, argv[2]=`world`

Argumente aus der Kommandozeile – Beispiel

```
#include <stdio.h>
main (int argc, char *argv[])
{
        while (– –argc>0)
                printf("%s%s",*++argv,(argc>1) ? " " : "");
                printf("\n");
        return 0;
}
```

- Programm aufrufen: `a.out hello world`
- Ausgabe: `hello world`

struct – Datenverbünde

```
struct Position {
  float x, y, z;
};
struct Particle {
  struct Position position;
  float ladung;
  int identitaet;
};
```

- **struct** definiert einen Verbund
- Ein Verbund fasst mehrere Variablen zusammen Die Datenverbünde können als eine Einheit verwendet werden
- Die Größe von Feldern in Verbünden muss konstant sein
- Ein Verbund kann Verbünde enthalten
- Verweis auf eine Komponente eines bestimmten Verbundes: Verbund-Variablenname.Komponente

Variablen mit einem struct-Typ

```
struct complex
   { double re; // 1. Komponente der Struktur
     double im; // 2. Komponente der Struktur
   } x,         // Variable x dieses Typs
     *cptr      //Zeiger auf eine Variable dieses Typs
  float f;      //reelle Variable
  f=x.re;       // Zugriff auf die 1. Komponente re von x
  f=cptr->im;   // Zugriff auf die 2. Komponente im von cptr
```

- Elemente des Verbunds werden durch „.“ angesprochen
- Kurzschreibweise für Zeiger: `(*pointer).x = pointer->x`
- Verbünde können wie Arrays initialisiert werden
- Initialisieren einzelner Elemente mit Punktnotation

Dateien - `fopen/fclose`

```
FILE *fopen (const char *Pfad, const char *Modus);
```

- `stdio.h` stellt Dateien durch *Filehandles* dar
- `Pfad`: Dateiname
- `fopen` gibt NULL zurück, wenn der Datenstrom nicht geöffnet werden konnte, ansonsten einen Zeiger vom Typ FILE auf den Datenstrom.
- Modus:
  - `r`: Datei nur zum Lesen öffnen (read)
  - `w`: Datei nur zum schreiben öffnen (write)
  - `a`: Daten an das Ende der Datei anhängen (append)
  - `r+`: Datei zum Lesen uns Schreiben öffnen (Datei muss schon existieren)
  - usw...
  - `b`: Binärmodus (anzuhängen an die obigen Modi, z.B. 'rb' oder 'r+b') (in unixoide Systemen hat es keine Auswirkungen)

fopen/fclose – Beispiel

```
#include <stdio.h>
int main (void)
{  FILE *datei;
  datei = fopen ("testdatei.txt", "w");
  fprintf (datei, "Hallo, Welt\n");
  fclose (datei);
  return 0;    }
```

- Dateien öffnen mit `fopen`, schließen mit `fclose`
- Alle nicht geschriebenen Daten des Stroms *datei werden gespeichert
- alle ungelesenen Eingabepuffer geleert
- der automatisch zugewiesene Puffer wird befreit
- der Datenstrom *datei geschlossen
- der Rückgabewert der Funktion ist EOF, falls Fehler aufgetreten sind, ansonsten ist er 0 (Null)

Sonstige I/O stdio.h Funktionen

```
fread  (void *daten, size_t groesse, size_t anzahl, FILE *datei
fwrite (const void *daten, size_t groesse, size_t anzahl, FILE
int fseek (FILE *datei, long offset, int von_wo);
```

- fprintf und fscanf funktionieren wie printf und scanf,
- aber auf beliebigen Dateien statt `stdout` (Bildschirm) und `stdin` (Tastatur)
- fread: liesst einen Datensatz
- fwrite: speichert einen Datensatz
- fwrite, fread Funktionen geben die Anzahl der geschriebenen/gelesenen Zeichen zurück
- die `groesse` ist jeweils die Größe eines einzelnen Datensatzes
- es können `anzahl` Datensätze auf einmal geschrieben werden.
- Sonst noch in `stdio.h`: fseek, fflush ...

stdio.h – sscanf

```c
#include <stdio.h>

int main(int argc, char **argv) {
  if (argc < 2) {
    fprintf(stderr, "Kein Parameter gegeben\n");
    return -1;
  }
  if (sscanf(argv[1], "%f", &fliess) == 1) {
    fprintf(stdout, "%f\n", fliess);
  }
  return 0;
}
```

- `sscanf` liest statt aus einer Datei aus einem 0-terminierten String
- Dies ist nützlich, um Argumente umzuwandeln
- Es gibt auch `sprintf` (gefährlich!) und `snprintf`

math.h – mathematische Funktionen

```
#include <math.h>

float pi = 2*asin(1);

for (float x = 0; x < 2*pi; x += 0.01) {
  printf("%f %f\n", x, pow(sin(x), 3)); // x, sin(x)^3
}
```

- `math.h` stellt mathematische Standardoperationen (z.B. die Wurzeln, Potenzen, Logarithmen, usw.) zur Verfügung
- Bibliothek einbinden mit
  `gcc -Wall -O3 -std=c99 -o mathe mathe.c -lm`
- Beispiel erstellt Tabelle mit Werten x und $sin(x)^3$

string.h – Stringfunktionen

```
#include <string.h>

char test[1024];
strcpy(test, "Hallo");
strcat(test, " Welt!");
// jetzt ist test = "Hallo Welt!"
if (strcmp(test, argv[1]) == 0)
  printf("%s = %s (%d Zeichen)\n", test, argv[1],
         strlen(test));
```

- `strlen(quelle)`: Länge eines 0-terminierten Strings
- `strcat(ziel, quelle)`: kopiert Zeichenketten
- `strcpy(ziel, quelle)`: kopiert eine Zeichenkette
- `strcmp(quelle1, quelle2)`: vergleicht zwei Zeichenketten, Ergebnis ist < 0, wenn lexikalisch `quelle1` $<$ `quelle2`, $= 0$, wenn gleich, und > 0 wenn `quelle1` $>$ `quelle2`

string.h – Stringfunktionen

```
#include <string.h>

char test[1024];
strncpy(test, "Hallo", 1024);
strncat(test, " Welt!", 1023 - strlen(test));
if (strncmp(test, argv[1], 2) == 0)
  printf("die 1. 2 Zeichen von %s und %s sind gleich\n",
    test, argv[1]);
```

- Zu allen `str`-Funktionen gibt es auch `strn`-Versionen
- Die `strn`-Versionen überprüfen auf Überläufe
- Die Größe des Zielbereichs muss *korrekt* angegeben werden
- Die terminierende 0 benötigt ein Extra-Zeichen!
- Die `str`-Versionen sind oft potentielle Sicherheitslücken!

Größe von Datentypen – sizeof

```
int x[10], *px = x;
printf("int: %ld int[10]: %ld int *: %ld\n",
        sizeof(int), sizeof(x), sizeof(px));
// → int: 4 int[10]: 40 int *: 8
```

- `sizeof(datentyp)` gibt an, wieviel Speicherplätze eine Variable vom Typ `datentyp` belegt
- `sizeof(variable)` gibt an, wieviel Speicherplätze die Variable `variable` belegt
- `sizeof` liefert einen ganzzahligen Wert ohne Vorzeichen zurück
- Bei Arrays: Länge multipliziert mit der Elementgröße
- Zeiger haben immer dieselbe Größe (8 Byte auf 64-Bit-Systemen)

Speicherverwaltung – malloc und free

```
#include <stdlib.h>
// Array mit Platz fuer 10000 integers
int *vek = (int *)malloc(10000*sizeof(int));
for(int i = 0; i < 10000; ++i) vek[i] = 0;
// Platz verdoppeln
vek = (int *)realloc(vek, 20000*sizeof(int));
for(int i = anzahl; i < 20000; ++i) vek[i] = 0;
free(vek);
```

- Speicherverwaltung für variabel große Bereiche
- `malloc` reserviert Speicher
- `realloc` verändert die Größe eines reservierten Bereichs
- `free` gibt einen Bereich wieder frei
- Wird dauernd Speicher belegt und nicht freigegeben, geht irgendwann der Speicher aus („Speicherleck“)
- Ein Bereich darf auch nur einmal freigegeben werden

typedef

```
typedef float length;
length len, maxlen;
length *vlength[];
```

- **typedef** definiert neue Namen für Datentypen
- **typedef** ist nützlich, um Datentypen auszutauschen, z.B. double anstatt float

```
typedef struct MyStruct {
    int data1;
    char data2;
} newtype;

newtype a;
```

- Deklarationen vereinfachen

const – unveränderbare Variablen

```
static const float pi = 3.14;

pi = 5; // Fehler, pi ist nicht schreibbar

// Funktion aendert nur, worauf ziel zeigt, nicht quelle
void strcpy(char *ziel, const char *quelle);
```

- Datentypen mit **`const`** sind konstant
- Variablen mit solchen Typen können nicht geändert werden
- Anders als Makros haben sie einen Typ
- Vermeidet seltsame Fehler, etwa wenn einem Zeiger ein **`float`**-Wert zugewiesen werden soll
- **`static`** ist nur bei Verwendung mehrerer Quelldateien wichtig