8. Speicherklassen, Verfügbarkeit und Lebensdauer

8.1. Verfügbarkeit (Scope) und Lebensdauer (Duration)

Für Namen (Variablen, Konstante, Funktionen) im Programm sind folgende beiden Gesichtspunkte von Wichtigkeit:

- **Verfügbarkeit (Scope):**
  Bereiche eines Programmes, in denen auf ein Objekt (Variable oder Funktion) zugegriffen werden kann. C unterscheidet vier Verfügbarkeitsbereiche:
  - Programm
  - Modul (Datei)
  - Funktion
  - Block (Verbundanweisung, { ... })

  Objekte, die im Programm (nicht in der Funktion `main()`!) oder einem Modul definiert werden, sind **globale** Objekte. Funktionen sind in C daher immer global.

  Objekte, die innerhalb eines Blocks oder einer Funktion definiert werden, sind **lokale** Objekte. Lokale Objekte können nur Variable sein.

- **Lebensdauer (Duration):**
  Dauer der Zuordnung von Speicherplatz zu einem Objekt. Man unterscheidet:
  - Gesamte Programmlaufzeit (static duration):
    Der Speicherbereich wird dem Objekt zu Programmbeginn zugeordnet. Die Zuordnung wird erst mit dem Programmende wieder aufgehoben.
  - Ausführungszeit des Blockes, in dem die Definition des Objektes erfolgte (automatic duration).
    Dem Objekt wird mit Eintritt des Programmes in den Block automatisch Speicher zugewiesen. Mit Verlassen des Blockes wird diese Speicherzuweisung wieder aufgehoben.

8.2. Speicherklassen

Speicherklassen können direkten Einfluss auf die Verfügbarkeit und die Lebensdauer von Objekten haben.

`auto`	"normale" lokale Variable (Standard-(default-)Speicherklasse, zufällig initialisiert) • alle Variablen, denen nicht explizit eine Speicherklasse zugewiesen wird und die nicht außerhalb von Funktionen vereinbart werden, fallen in die Speicherklasse auto • man kann einer Variablen explizit die Speicherklasse auto zuordnen, indem man vor die Typangabe bei der Variablenvereinbarung das Schlüsselwort auto setzt • automatische Variable werden bei jedem Funktionsaufruf neu

<table>
<tr><td></td><td>erzeugt und beim Verlassen der Funktion wieder zerstört
• daher ist ihr Geltungsbereich auf die lokale Funktion, in der sie vereinbart wurden, beschränkt
• dies gilt auch für Blöcke (Variablen, die innerhalb von Blöcken vereinbart werden und die in die Speicherklasse auto fallen, sind nur lokal in diesem Block bekannt)
• auto Variablen können beliebig (nicht nur mit Konstanten) initialisiert werden
• nicht initialisierte auto Variablen haben einen undefinierten Wert (es existiert kein Weglasswert!)
• auto Vektoren können nicht initialisiert werden

Beispiele:

<pre>int y = 1;   /* global */
int main(void)
  {
  int x = 5; /* lokal */

  printf("x=%d\n",x);   /* Ausgabe: 5 */
    {
    int x=10;
    printf("x=%d\n",x); /* Ausgabe: 10 */
    }
  printf("x=%d\n",x);   /* Ausgabe: 5 */
  }</pre></td></tr>
<tr><td>register</td><td>der "Wunsch", char- oder int-Variable im Maschinenregister zu speichern (zufällig initialisiert)

• in dieser Speicherklasse können sich einfache Variable befinden
• sie entspricht in ihren sonstigen Eigenschaften der Speicherklasse auto
• der Compiler versucht register Variablen so zu verwenden, daß sie in einem wirklichen Hardwareregister der CPU gehalten werden
• ist dies nicht möglich, so wird register ignoriert und die Variable wie auto behandelt</td></tr>
<tr><td>extern</td><td>lokale Deklaration von globalen Variablen (initialisiert mit 0)

• alle Objekte, die außerhalb von Funktionen vereinbart werden, sind in der Speicherklasse extern
• die Funktionsnamen selbst sind ebenfalls in der Speicherklasse extern
• externe Objekte sind ab ihrer Vereinbarung bis zum Ende des Quellencodes und sogar in anderen Quellencodedateien bekannt
• bei der Vereinbarung eines Objektes als extern kann es sich um eine Definition oder um eine Deklaration handeln
• die folgenden Vereinbarungen (immer außerhalb von Funktionen) unterscheiden sich wesentlich:
• <pre>int sp = 0;   /* Definition */</pre></td></tr>
</table>

<table>
<tr><td></td><td>

<code>double vector[N];</code>
<code>extern int sp; /* Deklaration */</code>
<code>extern double vector[];</code>
bei einer Definition wird eine Variable erzeugt
nur hier ist eine Initialisierung möglich
fehlt eine Initialisierung, so wird der Weglasswert 0 eingesetzt
bei einer Deklaration werden nur die Variableneigenschaften festgelegt, die Variable selbst muß an anderer Stelle im Quellencode definiert sein
Deklarationen dürfen für eine bestimmte Variable mehrmals im (gesamten) Quellencode vorkommen, eine Definition aber nur einmal

Beispiel: Text einlesen und die Häufigkeit der Zeichen zählen
<pre>
#include <stdio.h>
#define MAXCHARS 255

int count[MAXCHARS];   /* global */

void print_alle (void);

void main(void)
  {
  extern int count[];   /* Deklaration von count */
  int c;
  while((c=getchar()) != EOF)
    count[(char) c]++ ;
  print_alle();
  }

void print_alle(void)   /* kann z. B. in einer
anderen Datei stehen */
  {
  extern int count[];   /* Deklaration von count */
  int i;

  for(i = 0; i < MAXCHARS; i++)
    printf("%d: %d\n",i,count[i]);
  }
</pre>
</td></tr>
<tr><td><code>static</code></td><td>"statische" Variable, Inhalt bleibt erhalten (initialisiert mit 0)

static Objekte können sowohl intern als auch extern sein
static Objekte innerhalb von Funktionen sind nur lokal bekannt, behalten im Gegensatz zu auto Objekten aber ihre Werte zwischen den Funktionsaufrufen bei
bzgl. der Initialisierung gilt dasselbe wie für externe Objekte, static Vektoren sind daher initialisierbar
Zeichenketten innerhalb von Funktionen sind immer in der Speicherklasse static (z.B. printf() Parameterstring)
static Objekte außerhalb von Funktionen sind externe Objekte, deren Namen aber nur in dieser Quellencodedatei bekannt ist.

</td></tr>
</table>