Deklarationen in C

Prof. Dr. Margarita Esponda

Deklarationen

Deklarationen spielen eine zentrale Rolle in der C-Programmiersprache.

Deklarationen
- Variablen
- Funktionen

Die Deklarationen von Variablen und Funktionen haben viele Gemeinsamkeiten. Je nachdem wie und wo die Deklarationen gemacht werden, gelten Konzepte wie:

Lebensdauer

Gültigkeitsbereich

Verknüpfungen

Speicherung

Syntax der Deklarationen

Eine Deklaration liefert dem Compiler die nötige Information über die Variablen und Funktionen.

Die Variablen-Deklaration

```
float a;
```

sagt dem Compiler, dass in der aktuellen Umgebung (Programmblock) die Variable mit Namen a den Datentyp float hat.

Die Funktions-Deklaration

```
float f ( float a );
```

oder

```
float f ( float );
```

sagt dem Compiler, dass f eine Funktion ist, die als Rückgabewert einen Wert vom Typ float liefert und als Argument einen float Datentyp braucht.

Syntax der Deklarationen

Die allgemeine Syntax für Variablen- und Funktionsdeklarationen sieht wie folgt aus:

declaration-specifiers-sequence

Hier werden die Eigenschaften der deklarierten Variablen und Funktionen spezifiziert

declarators

Hier sind die Namen der deklarierten Variablen und Funktionen und Namen weiterer Eigenschaften, über die diese hier spezifiziert werden

Syntax der Deklarationen

Es gibt drei Sorten von Deklarationsspezifizierern

Deklarationsspezifizierer

declaration-specifiers-sequence

- Speicherklassen
  - **auto**
  - **static**
  - **extern**
  - **register**
- Datentyp-Spezifizierer
  - **void, char, short, int, long, float, double, signed, unsigned**
- Datentyp-Qualifizierer
  - **const**
  - **volatile**

Syntax der Deklarationen

Beispiele:

```
static const unsigned long int number [ 25 ];
```

- **static** → Speicherklasse
- **const** → Qualifizierer
- **unsigned long int** → Datentyp-Spezifizierer
- **number [25]** → Deklarator

```
extern unsigned long int square ( int );
```

- **extern** → Speicherklasse
- **unsigned long int** → Datentyp-Spezifizierer
- **square (int)** → Deklarator

Eigenschaften von Variablen

Jede Variable in C hat drei wichtige Eigenschaften:

Lebensdauer: ab wann und wie lange eine Variable während der Ausführung eines Programms im Speicher existiert

Gültigkeitsbereich: für welchen Teil des Programms eine Variable sichtbar ist

Verbindungen (Verknüpfungen) inwieweit eine Variable von verschiedenen Programmteilen oder Modulen gemeinsam benutzt werden kann

Speicherklassen

Es gibt vier Sorten von Speicherklassen:

auto

static

extern

register

Es darf maximal eine Sorte von Speicherklassen in einer Deklaration vorkommen.

Wir beschränken uns zunächst einmal auf Variablendeklarationen.

Speicherklassen

auto

Alle Variablen, die nicht explizit mit einer Speicherklasse und nicht außerhalb von Funktionen deklariert worden sind, haben automatisch die Speicherklasse **auto.**

auto-Variablen werden bei jedem Funktionsaufruf neu erzeugt und beim Verlassen der Funktion wieder zerstört.

Ihr Geltungsbereich beschränkt sich auf die Funktion, in dem sie vereinbart worden ist.

Nicht initialisierte **auto**-Variablen haben einen undefinierten Wert.

Speicherklassen

auto

Beispiel:

```
void  func ( void )
{
  Artikel  a;
  . . .
}
```

auto-Speicherklasse

Gültigkeitsbereich nur innerhalb der Funktionsdefinition

keine Verknüpfung zur Außenwelt

Speicherklassen

register

Vermittelt den Wunsch zum Compiler, die Variable im Maschinenregister zu speichern anstatt im Hauptspeicher. Der Compiler entscheidet selber, ob er das tut oder nicht.

Nur Variablen mit Datentyp **char** oder **int** können als **register** deklariert werden.

Ursprünglich haben die Compiler versucht, diese Variablen so zu verwenden, dass sie in einem wirklichen Hardwareregister der CPU gehalten werden

Moderne Compiler sind durch die Entstehung hoch paralleler Prozessoren komplizierter geworden. Meistens entscheiden die Compiler selber, welche Variablen am besten im Register gehalten werden und welche nicht, um die Ausführung des Programms zu beschleunigen.

static

Wenn eine Variable **innerhalb einer Funktion** als **static** deklariert wird, hat sie einen Gültigkeitsbereich nur innerhalb der Funktion, aber **erhält ihren Wert zwischen den Funktionsaufrufen.**

static-Variablen **außerhalb von Funktionen** sind Variablen, die nur **innerhalb der Quellcodedatei bekannt** sind.

Die Variablen sind außerhalb der Quellecodedatei unsichtbar, und nur Funktionen innerhalb der Datei können die Variablen teilen und gemeinsam benutzen.

static

```
static char c;
```

Gültigkeitsbereich nur innerhalb der Quellcodedatei

Lebensdauer solange das Programm ausgeführt wird

interne Verknüpfung nur beschränkt auf Funktionen der Quellcodedatei.

```
void func ( void )
{
    static char p;
}
```

Gültigkeitsbereich nur innerhalb der Funktion

Lebensdauer solange das Programm ausgeführt wird

keine Verknüpfung

static

Statische Variablen können **für die Kapselung von Information sehr nützlich w**erden.

Variablen, die als **static** deklariert worden sind, werden vor der Programmausführung **nur einmal initialisiert.**

Wenn eine **static**-Variable innerhalb einer Funktion deklariert wird, gibt es eine noch strengere Kapselung, weil die Variable nur innerhalb der Funktion zugreifbar ist.

extern

Eine externe Variablendeklaration erlaubt die **gemeinsame Verwendung** der Variable **innerhalb mehrerer Quellcode-Dateien**.

Alle Variablen, die außerhalb von Funktionen vereinbart werden, gehören automatisch zur Speicherklasse extern.

```
extern int i;
```

Informiert nur den Compiler, dass **i** eine Variable vom Typ **int** ist, aber es wird kein Speicherplatz für diese Variable reserviert. Es wird nur dem Compiler gesagt, dass wir Zugriff auf die Variable haben möchten und das eventuell später im Programm oder in einer anderen Datei die Variable definiert wird.

extern

```
extern int i;
```

In die **C**-Terminologie ist das nur eine Deklaration, aber keine Definition.

Eine Definition findet statt, wenn die Variable auch gleichzeitig initialisiert wird.

Eine als extern deklarierte Variable kann mehrmals im Programm als solche deklariert werden, aber nur einmal definiert werden.

Diese Regelung verhindert, dass eine externe Variable mehrmals initialisiert wird.

extern

```
extern char c;
```

Gültigkeitsbereich nur innerhalb der Quellcodedatei

Lebensdauer solange das Programm ausgeführt wird

? Verknüpfung

```
void func ( void )
{
    extern char p;
}
```

Gültigkeitsbereich nur innerhalb der Funktion

Lebensdauer solange das Programm ausgeführt wird

? Verknüpfung

Speicherklassen mit Funktionen

Funktionen können nur die Speicherklassen static und extern haben.

Eine Funktion, die als extern deklariert worden ist, kann aus anderen Quelldateien ausgeführt werden.

Eine Funktion, die als static deklariert wird, kann nur von Funktionen aufgerufen werden, die innerhalb der selben Quellcodedatei sind.

```
extern int  f ( void ) ;
static int  g ( void ) ;
int  h ( void ) ;
```

externe Verbindung

interne Verbindung

externe Verbindung (default-mäßig)

Qualifizierer

Es gibt nur zwei Qualifizierer **const** und **volatile**

Der **volatile**-Qualifizierer wir nur bei Hardware naher Programmierung verwendet.

Der **const** Qualifizierer definiert Variablen als nur lesbar (read-only).

```
const float PI = 3.14589;
```

Beim ersten Blick könnte man denken, dass eine **const**-Deklaration dasselbe ist wie **define**-Direktive. Es gibt aber wesentliche Unterschiede zwischen beiden.

Qualifizierer

Unterschied zwischen einer **define**-Direktive und const

Die **define**-Direktive kann nur für Zahlen-, Zeichen- und String-Konstanten verwendet werden, während **const** mit einem beliebigen Datentyp benutzt werden kann.

Die mit **const** deklarierten Variablen können verschiedene Gültigkeitsbereiche haben, mit **define**-Direktiven deklarierte Variablen nicht. Insbesondere können wir nicht **define**-Variablen mit lokalem Gültigkeitsbereich haben.

Eine mit **const** deklarierte Variable kann mit einem debugger gesehen werden, mit **define**-Direktiven deklarierte Variablen nicht.

Der einzige Nachteil von **const** ist, dass es nicht in konstanten Ausdrücken verwendet werden kann.

Beispiel:

```
const int n = 10;
int   a[n];
```

Deklaratoren

Ein Deklarator besteht aus einem Bezeichner, der der Name einer Variable oder einer Funktion ist.

Das *****-Symbol kann vorstehen, und **[]** oder **()** können dem Bezeichner folgen.

```
int  i;
int  *i;
int  a[15];
int  func();
```

Deklarationen von Arrays mit leeren Zeichen können nur gemacht werden, wenn externe Variablen, Parameter von Funktionen oder die Initialisierung später gemacht wird.

```
extern  int  a[];
```

Deklaratoren

Einfache Deklarationen, die uns bekannt sind.

```
int*   z_p[100];
float*  func ( float  num );
float*  func ( float );
```

```
int   *z_p[100];
float  *func ( float  num );
float   *func ( float );
```

Deklarationen mit Zeiger auf Funktionen.

```
void (*func) (int);
```

Deklaration von einem Zeiger auf eine Funktion mit einem Parameter vom Typ int und einem Rückgabewert von Typ void.

Deklaratoren

Kompliziertere Deklaratoren:

```
int *(*f[100]) (void)
```

Zwei einfache Regeln, um solche Deklarationen zu entziffern:

Zuerst den Bezeichner suchen

Eckige Klammern und runde Klammern gegenüber * bevorzugen. D.h., wenn ein Stern vor dem Bezeichner steht und danach eckige Klammern, dann heißt dies, dass wir zuerst ein array haben und nicht einen Zeiger, und wenn wir runde Klammern haben, dann ist das eine Funktion und nicht ein Zeiger.

Initialisierer

```
int i = 2;
int j = 0.5;
```

Zeigervariablen

```
int *p = &i;                /*  i muss den Datentyp int haben */
int a[5] = {1, 0, 1, 1, 0};
```

Hier können nur konstante Werte stehen

Statische und konstante Variablen können nur mit Konstanten initialisiert werden, auto-Variablen auch mit nicht-konstanten Werten.

```
#define   MAX  100

static  int   i = MAX;
```

```
int func ( int n )  {
    int i = n;
}
```

Initialisierer

Variablen mit auto-Speicherklasse haben einen undefinierten Wert, wenn wir diesen nicht explizit im Programm initialisieren.

Variablen mit static-Speicherklasse werden mit dem Wert **0** initialisiert. Anders als mit der Funktion calloc, die einfach alles mit null-Bits initialisiert, werden static-Variablen je nach ihrem Datentyp mit **0**, **0.0** oder mit der Konstante **null** initialisiert.

Eine gute Programmierdisziplin ist es, alle Variablen explizit im Programm zu initialisieren.

Zeiger auf Funktionen

Funktionen befinden sich auch im Speicher, besitzen also auch Adressen.

In C ist ein Funktionsname genauso wie in Vektornamen eine Adresskonstante.

Es können Zeiger auf Funktionen definiert werden, die dann wie Zeigervariablen verwenden werden können.

Damit wird es möglich, Funktionen an Funktionen als Parameter zu übergeben.

Zeiger auf Funktionen

Bei der Deklaration einer Zeigervariablen auf eine Funktion muss deren Parameterliste und Rückgabetyp angegeben werden.

Beispiel:

```
...
  int n;
  int  (*func_pointer) (double, int);
  func_pointer = eigeneFunktion;
  n = func_pointer (2.9, 3);
...
```

Zeiger auf Funktionen

Beispiel:

```
#include <stdio.h>

int funk1 (void)  {
     printf ( "Ich bin in funk1 ... " );
     return 7;
}

int halbiert (int n) {
     printf ( "Ich bin in halbiert ... " );
     return n/2;
}

int main (void)  {
      int (*fp1)(void);
      int (*fp2)(int);
      int n;
      fp1 = funk1;
      n = fp1();       // Aufruf über den Zeiger
      printf( "%d" ,n );
      fp2 = halbiert;
      n = fp2( n );   // Aufruf über den Zeiger
      printf( "%d" ,n );
      return 0;
}
```

Funktionen als Parameter von Funktionen

Beispiel:

```
double integrate( double (*f)(double), double a, double b);
```

```
double integrate( double f(double), double a, double b);
```

gleiche Semantik

Verwendungsbeispiel der Funktion innerhalb der Integrate-Funktion

```
...
  summe += (*f)(x);
...
```

C99 Standard

Wichtige Erneuerungen

- Datentypen

  long long int

  unsigned long long int (mindestens 64 Bit groß)

  _Bool

  _Complex

  bool (mit #include <stdbool.h>)

- Lokale Felder mit variabler Größe

- Frei platzierbare Deklaration von Bezeichnern

  (in C90 durften diese nur am Anfang eines Blocks stehen)

- Alternative Schreibweisen für Operatoren:

  Beispiele: **and** anstelle von **&&**

  or anstelle von **||**

- Erweiterte Bibliotheken (z.B. complex.h, math.h)

Formatierung von Funktionen aus GNU

Funktionsname an der ersten Spalte

```
static char *
concat (char *s1, char *s2)
{
  ...
}
```

Wenn die Argumente nicht in eine Linie zusammen passen.

```
...
int
lots_of_args ( int an_integer, long a_long, short a_short,
               double a_double, float a_float )
...
```

Formatierung von Kontrollstrukturen aus GNU

```
if (x < foo (y, z))
  haha = bar[4] + 5;
else
  {
    while (z)
      {
        haha += foo (z, z);
        z--;
      }
    return ++x + bar ();
  }
```

```
if (foo)
  ...
else
  {
    if (bar)
      ...
  }
```

```
do
  {
    a = foo (a);
  }
while (a > 0);
```

```
if (foo_this_is_long && bar > win (x, y, z)
    && remaining_condition)
```