Programmierung in C: Vermischtes (Teil 1)

- **Gültigkeitsbereiche für Variablen**
- C-Präprozessor (#define EPS 0.00001)
- Inkremtieren und Dekrementieren (jahr++; restzeit--;)
- Speicherklassen (static)
- Typecasts

Teil 2 folgt später

- Fehlersuche in Programen
- Organisation des Quellcodes

Gültigkeitsbereiche (1)

Variablen können verschiedene Gültigkeitsbereiche besitzen:

lokale Gültigkeit und globale Gültigkeit

Lokale Gültigkeit

Variablen können nur in der Funktion benutzt werden, in der sie deklariert werden. Eine lokale Gültigkeit kann auf die main-Funktion, aber auf jede andere Funktion, oder auch auf Verbundanweisungen bezogen sein.

Lokale Gültigkeit innerhalb der main()-Funktion: Die Variablen sind nur innerhalb der main-Funktion sichtbar. Andere Funktionen (Prozeduren) können nicht direkt auf diese Variablen zugreifen. Bei Bedarf muss der Wert der Variablen den Funktionen über Parameter vermittelt werden.

Gültigkeitsbereiche (2)

Beispiel zur lokalen Gültigkeit innerhalb der main()-Funktion:

```
int berechne(int arg)
   {
     // b, c sind hier nicht sichtbar, a wird über arg vermittelt

     return arg*2;
   }

main()
   {
     int a, b, c;
     …
     b = berechne(a);
     …
   }
```

Gültigkeitsbereiche (3)

lokale Gültigkeit in einer Funktion (Prozedur): Nur der Code innerhalb der Funktion kann diese Variable benutzen. Wird aus der Funktion zurückgesprungen, sind die Variablen nicht mehr sichtbar. Ihr Speicherplatz wird freigegeben und für andere Zwecke verwendet.

```
double fakultaet(int arg) {
  double f=1.0;
  int i;  /* ist nur lokal gültig */
  if (arg<2) return 1.0;
  for (i=2;i<=arg;i++) f = f * (double)i;
  return f;
}

main()
{
  double y = fakultaet(42);
  printf("%lf ", y);
  printf("%d",i);  /* Fehler: i ist hier nicht sichtbar */
}
```

Gültigkeitsbereiche (4)

lokale Gültigkeit in einer Funktion (Prozedur) …

Achtung: Bei mehrmaligem Aufruf einer Funktion bleibt der Wert einer lokalen Variablen i.A. nicht gespeichert (Ausnahme durch spezielle Speicherklasse). Die Variable verliert zwischenzeitlich ihre Gültigkeit und wird bei jedem neuen Eintritt in die Funktion neu erzeugt.

Sichtbarkeit und Überdeckung lokaler Variable:

```
void f1(int *a) { int x=5; *a = *a+f2(&x); }

void f2(int *a) { int x=77; *a = *a+x; }
```

Die Variable x in *f2* überdeckt x aus *f1*. Innerhalb *f2* wird mit dem Wert x=77 gearbeitet. Die Variable x innerhalb *f1* bleibt bestehen, ist aber nicht sichtbar, wenn *f2* abgearbeitet wird

Gültigkeitsbereiche (5)

Globale Gültigkeit:

Globale Variablen sind über alle Funktionen hinweg gültig. Eine Ausnahme ist die lokale Überdeckung durch gleich benannte Variablen in Funktionen oder Verbundanweisungen.

Durch globale Variable können Daten zu und von Funktionen vermittelt werden:

Vorteile: bequem für Programmierer;
schnell, da kein Kopieraufwand bei Aufruf der Funktion

Nachteil: Die Funktionsdeklaration enthält keine Information, welche globalen Daten zur Eingabe und/oder Ausgabe benutzt werden.
Man verliert schnell den Überblick und kann nicht mehr einschätzen, ob der Aufruf von Funktionen eventuell unerwünschte Nebenwirkungen hat.

→ Globale Variablen sehr sparsam verwenden!

Gültigkeitsbereiche (6)

Beispiel für globale Gültigkeit:

```
int vermoegen=0; // globale Variable

void erbschaft( int betrag)
   { vermoegen = vermoegen + betrag;
     zahle_erb_steuer(betrag);
   }

void zahle_erb_steuer(int erb_betrag)
   { vermoegen = vermoegen
        – (int)( (float)erb_betrag * (float) E_STEUERSATZ/100.0);
   }

main()
   { ...
     vermoegen = X;
     erbschaft(Y);
      if (vermoegen >= 1000000) printf(“ Ich bin reich! \n“);
   }
```

Programmierung in C: Vermischtes (Teil 1)

- Gültigkeitsbereiche für Variablen
- **C-Präprozessor (#define EPS 0.00001)**
- Inkremtieren und Dekrementieren (jahr++; restzeit--;)
- Speicherklassen (static)
- Typecasts

Präprozessor

Vor dem Übersetzen des Programm wird der s.g. Präprozessor (engl. preprocessor) ausgeführt.

Aufgaben:

- Automatisiertes Einfügen von externen Dateiinhalten in C-Programme, zum Beispiel *#include <stdio.h>*
- Ersetzungen für definierte Konstanten und Makros, zum Beispiel *#define MAX_BENUTZER 100*
- Einbeziehen von ausgewählten Programmteilen in die Übersetzung, Ausblenden anderer Programmteile

Präprozessor-Konstanten (1)

Für Parameter, die zur Übersetzungszeit festgelegt werden, bieten sich Präprozessorkonstanten an.

Allgemeine Form:
#define NAME WERT

- NAME ist der Konstantenname, der im Programm benutzt wird
- WERT ist dabei eine Zeichenkette, die aber auch einen Konstantenausdruck repräsentieren kann.

Vor den eigentlichen Übersetzen wertet der Präprozessor die #define Anweisungen aus, und setzt überall dort, wo NAME als Bezeichner im Quellcode steht, die Zeichenkette WERT ein.

Präprozessor-Konstanten (2)

Beispiel:
Maximale Anzahl von Pixeln (hier 4000x4000) in einem Bild, gesteuert durch Präprozessorkonstanten

```
#define MAX_HORIZ 4000
#define MAX_VERT 4000

typedef struct { char r,g,b;} pixel;

pixel bild[MAX_VERT][MAX_HORIZ];

for(int v=0; v<MAX_VERT; v++)
   for(int h=0; h<MAX_HORIZ; h++)
   {   bild[v][h].r = 100;
       bild[v][h].g= 100;
       bild[v][h].b= 50;
   }
```

Präprozessor-Makros (1)

Der Präprozessor kann auch kleine Makro-Funktionen mit Parametern in dem Code ersetzen.

```
#define ABS(x) ((x)<0?-(x):(x))

….
float a,b;
int x,y;

….
b = ABS(a);  // wird ersetzt durch: b = a<0?-a:a;
y = ABS(x);  // wird ersetzt durch: y = x<0?-x:x;
```

Vorteil: Man kann Details in Funktionen verbergen. Der Aufwand eines Funktionsaufrufs wird aber zur Laufzeit vermieden. Programme laufen dadurch schneller!

Präprozessor-Makros (2)

Unterschied von Makros gegenüber echten C-Funktionen:

- Makro-Funktionen werden durch Ersetzung im Quelltext realisiert.
- C-Funktionen werden zur Laufzeit aufgerufen und als ganzes ausgeführt.

Wenn Operationen innerhalb und außerhalb der Makros/Funktionen verschiedene Prioritäten aufweisen, kann sich ein unterschiedliches Verhalten ergeben, je nachdem ob man eine Makro-Funktion oder eine echte C-Funktion benutzt.

Beispiel:

```
#define F(x,y) (x)+(y)
…
a=1;
b=2;
c=3;

d=F(a,b)*c;  // d = a+b*c;
             // d = 1+2*3 =
```

```
int f(int x, int y) { return x+y; }
…
a=1;
b=2;
c=3;

d1=f(a,b)*c;  // f(1,2) = 3
              // d = 3* c = 9
```

Präprozessor-Anweisungen

Präprozessor-Anweisungen können zur Versionierung des Codes benutzt werden

```
#ifdef  VERSION_1
  int a = 0,b = 0, c =0; // hier alles mit Integerzahlen
#endif
#ifdef VERSION_2;
  float a = 0.0, b = 0.0, c = 0.0; // jetzt alles mit Fließkommazahlen
#endif
```

Präprozessor-Anweisungen auch zum “Auskommentieren“ von Code – anstelle geschachtelter Kommentare

```
#ifdef IRGENDETWAS_UNDEFINIERTES
    int anzahl = 0;
    int i;
    …
#endif
```

Programmierung in C: Vermischtes (Teil 1)

- Gültigkeitsbereiche für Variablen
- C-Präprozessor (#define EPS 0.00001)
- **Inkremtieren und Dekrementieren (jahr++; restzeit--;)**
- Speicherklassen (static)
- Typecasts

Spezielle Operatoren für Inkrement und Dekrement

Inkrement heißt „um Eins hochsetzen“, Dekrement „um Eins erniedrigen“

zahl = zahl + 1; // kann verkürzt werden zu
zahl += 1; // und nochmals zu
zahl++;

Entsprechend gibt es zahl--; für das Erniedrigen von zahl um 1.
Beachte:
„zahl++;“ und „++zahl;“ sind nicht unbedingt das gleiche.

Beispiel:
int zahl1 = 64;
int zahl2 = ++zahl1; // zahl2 und zahl1 sind nun beide 65
int zahl3 = zahl2++; // zahl3 ist 65 und zahl2 66.

Spezielle Operatoren für Inkrement und Dekrement

Die Operatoren können als Prefix- oder als Postfix-Variante verwendet werden.

Prefix-Inkrement:

```
int zahl2 = ++zahl1;

// entspricht
int zahl2 =PreInc(zahl1);

// mit
int PreInc(int& i)
{
  i = i + 1;
  return i;
}
```

Postfix-Inkrement:

```
int zahl2 = zahl1++;

// entspricht
int zahl2 =PostInc(zahl1);

// mit
int PostInc(int& i)
{
  int temp=i;
  i = i + 1;
  return temp;
}
```

Programmierung in C: Vermischtes (Teil 1)

- Gültigkeitsbereiche für Variablen
- C-Präprozessor (#define EPS 0.00001)
- Inkremtieren und Dekrementieren (jahr++; restzeit--;)
- **Speicherklassen (static)**
- Typecasts

Speicherklassen (1)

C besitzt vier Schlüsselworte, um die Speicherklasse von Variablen zu definieren. Sie teilen dem Übersetzer mit, wie eine Variable zu speichern ist.

auto – Standard für lokale Variablen
register – wie auto, Variable wird vorrangig in einem Prozessorregister gespeichert (anstatt im Hauptspeicher)
…
auto und *register* nur für lokale Variablen

static – Variable behält den Wert auch zwischen Funktionsaufrufen
extern – Variable ist bereits an anderer Stelle deklariert und soll nur als Bezeichner noch einmal eingeführt werden … sinnvoll bei mehreren Quelletextdateien.

Speicherklassen (2)

Beispiel für static:

```
int zaehlen()
{ static int zahl=0;
  zahl = zahl+1;
  return zahl;
}

main()
{
  int x, ergebnis;
  for ( x=1;x<47;x=x+1)
    ergebnis = zaehlen();
  printf("Das Ergebnis lautet %d\n", ergebnis);
}
```

Programmierung in C: Vermischtes (Teil 1)

- Gültigkeitsbereiche für Variablen
- C-Präprozessor (#define EPS 0.00001)
- Inkremtieren und Dekrementieren (jahr++; restzeit--;)
- Speicherklassen (static)
- **Typecasts**

Typecasts

Der Übersetzer wandelt Typen von einem in einen anderen um, wenn es notwendig wird. In einigen Fällen werden Warnungen generiert, falls es zu Genauigkeitsverlust kommt oder Zuweisungen mit unterschiedlichem Wertebereich angewiesen werden.

Man kann durch Typcasts aber auch eine Typumwandlung explizit anweisen.

Typischer Fall:

```
int a=3,b=5;
float x_verhaeltnis_ab = a/b; /* Ergebnis int, 0 */
float y_verhaeltnis_ab = (float) a/b; /* Ergebnis float, 0.0 */
float z_verhaeltnis_ab = (float) a/(float) b; /*ergibt 0.6 */
```