Hochschule **Rosenheim**
University of Applied Sciences

Prozedurale Programmierung

Einfache Zeiger

Hochschule Rosenheim - University of Applied Sciences

WS 2018/19

Prof. Dr. F.J. Schmitt

Erinnerung: Funktionen

- Prinzipiell gilt:
  - Bei einem Funktionsaufruf werden in C die Werte an die Funktionsparameter immer in Form von Kopien (call by value) und nicht im Original (call by reference) übergeben

```
long Erhoehe(long a)
{
   a = a+1;
   return a;
}
int main(void)
{
   long y, x=1;
   y = Erhoehe(x);
   return 0;
}
```

Änderung von a in der Funktion hat keine Wirkung nach außen!

⇒ was tun, wenn man das aber möchte?

Problem: Mehr als ein Rückgabewert

- C erlaubt nur einen einzigen Rückgabewert in Funktionen
- was tun, wenn man mehrere möchte?
- Beispiel: Vertauschung von Variableninhalten

Problem: Große Datenmengen (struct)

- struct-Daten werden wie normale Datentypen behandelt
- und bei der Übergabe an Funktionen komplett kopiert
- Problem:
  - Datenmenge kann je nach struct-Definition sehr groß sein
  - kopieren ist relativ langsam
  - und möglicherweise unnötig:
    - call-by-value stellt sicher, dass Daten nicht verändert werden
    - evtl. will man das ja gar nicht → trotzdem wird kopiert

Lösung

- Übergabe von Zeigern mit call-by-value

- Effekt
  - Änderungen können nach außen wirksam werden
  - es wird keine Kopie der angesprochenen Variablen übergeben

nochmal: Erhoehe

```
void Erhoehe(long *a)
{
    *a = *a + 1;
}

int main(void)
{
    long x = 1;
    Erhoehe(&x);

    return 0;
}
```

Änderung von *a in der Funktion hat Wirkung nach außen!

x hat nach Aufruf den Wert 2

statt Kopie der Variablen → Adresse der Variablen im Speicher (Zeiger)

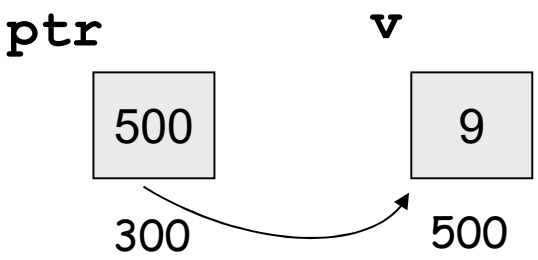
Prinzip

- Zeiger ist eine Variable, die eine Adresse enthält
- Speicherorganisation:

  - Irgendwo im Hauptspeicher befinden sich die beiden Variablen `ptr` und `v`
  - Variable `ptr` ist ein Zeiger
  - Variable `v` ist vom Typ `long`

Prinzip

- Angenommen:
  - `ptr` enthält die Adresse von `v`
  - `v` enthält den Wert `9`



- Fragen:
  - woher weiß man, dass die Variable eine Adresse enthält?
  - woher weiß man an welcher Adresse eine Variable im Speicher steht?

Definition

- Definition eines Zeigers:

```
Typ *Zeigername;
```

- Beispiel:

```
long *ptr;
```

  - Zeiger haben wie alle Variablen einen Datentyp
  - Datentyp von `ptr` ist Zeiger auf `long`
  - * gibt an, dass die Variable ein Zeiger ist und eine Adresse enthält

Adresse einer Variablen

- Adressoperator **&**
  - Unärer Operator
  - Liefert die Adresse einer Variablen (oder eines Lvalues)

```
long v = 9;
long *ptr;

ptr = &v;
```

  - Zuweisung der gelieferten Adresse an `ptr`
  - „ptr zeigt auf v“
  - Kann nicht auf Konstanten angewendet werden

Inhalt der Adresse

Inhaltsoperator *

- Unärer Operator
- Wird verwendet, wenn man das Objekt erreichen will, auf das ein Zeiger zeigt

```
long v = 9;
long *ptr;

ptr = &v;
*ptr = 3;
```

- Wert der Variablen `v` wird neu gesetzt
- „Dem Objekt, auf das `ptr` zeigt, wird 3 zugewiesen“

Operatoren & und *

- Wird einem Zeiger der Operator * vorangestellt, so wird dieser dereferenziert
  (⇒ man erhält das Objekt, auf das er zeigt)

- Operatoren & und * kehren einander um
  - `*&v` ist äquivalent zu `v`
  - `&*ptr` ist äquivalent zu `ptr`

- Verwechselungsgefahr
  - * in Variablendefinition: sagt dem Compiler, dass die Variable ein Zeiger ist
  - * sonst als unärer Operator: Dereferenzierung

NULL-Zeiger

- Zeigervariablen müssen initialisiert werden
  - Nicht initialisierte Zeiger enthalten irgendeine willkürliche Adresse
  - Beim Zugriff auf das Objekt durch Dereferenzieren des Zeigers wird auf falschen Speicherbereich zugegriffen (Programmabsturz)
- Manchmal wichtig Zeiger als ungültig zu markieren
  - D.h. er zeigt auf kein Objekt
  - Kann noch nicht benutzt werden
  - Zeiger wird hierfür mit Adresse 0 initialisiert
  - hierfür wird in C die Konstante NULL bereitgestellt
- Vergleich mit NULL gibt an ob dieser auf gültige Daten zeigt

Beispiel

```
long feld;
long *ptr = &feld; //ptr gesetzt

// ptr darf verwendet werden
//...

ptr = NULL;  //ptr ist ungültig
// ptr darf nicht mehr verwendet werden
```

Zuweisungen

- Wertzuweisungen zwischen Zeigern möglich

```
long a;
long *ap, *ptr;

ap = &a;
ptr = ap;
```

  - Wichtig: Beide Zeiger müssen den selben Datentyp haben !
  - Beide Zeiger zeigen anschließend auf die selbe Variable a

Zeiger als Parameter (1)

- Funktionen haben in C nur einen Rückgabewert
- Was wird gemacht wenn eine Funktion mehr als nur einen Wert an den Aufrufer zurückgeben soll?
  - Zeiger müssen als Parameter der Funktion verwendet werden

```
void SwapLong(long *x, long *y)
{
  long h;

  // Dreieckstausch
  h = *x;
  *x = *y;
  *y = h;
}
```

Zeiger als Parameter (2)

- Möglicher Aufruf der Funktion `SwapLong`:

```
...
long a,b;
//...
SwapLong(&a,&b);
```

  - Es werden die Adressen der beiden Variablen a und b übergeben und deren Werte werden vertauscht
  - Wären die Parameter x und y „reguläre“ Variablen
    - würden nur Kopien der Werte von a und b übergeben
    - Tauschen bliebe wirkungslos, da nur Kopien getauscht werden

Aufgabe

- Schreiben Sie eine Funktion, die
  - Breite und Höhe eines Rechtecks übergeben bekommt
  - Fläche und Umfang des Rechtecks zurückliefert

Funktionen und Strukturen

- Strukturen können im Gegensatz zu elementaren Datentypen sehr groß werden
- Übergabe als Kopie (call-by-value) ist zeitaufwändig

- Empfehlung:
  - Bei zeitkritischen Funktionen bereits ab einer Strukturgröße von 4 bis 8 Byte nicht die Kopien von Strukturen, sondern ihre Adressen als Zeiger übergeben
  - Aufwändiges Kopieren wird vermieden und statt dessen werden nur Zeiger (Adresswerte) kopiert

Erinnerung: Beispiel struct

- Variablendefinition

```
struct Punkt_s p1;
```

```
struct Punkt_s
{
  double x;
  double y;
};
```

- Zugriff auf Attribute mit dem Punkt-Operator (Selektionsoperator)

```
p1.x = 3.6;
p1.y = 4.3;
```

Zeiger auf Strukturen (1)

```
struct Punkt_s p1;
p1.x = 3.6;     p1.y = 4.3;
struct Punkt_s *pp = &p1;
```

Zeiger auf eine Struktur

- Zugriff auf eine Komponente

```
(*pp).x
```

  - Klammerung ist notwendig, da Selektionsoperator höhere Priorität hat als der Operator *

- Vereinfachte Schreibweise:

```
pp->x   anstatt   (*pp).x
```

„Hole das Objekt `x` aus dem Objekt, auf das `pp` zeigt."

Zeiger auf Strukturen (2)

```
void neuesRechteck(const struct Punkt_s *p1,
                   const struct Punkt_s *p2,
                   struct Rechteck_s *rect )
{
  rect->pmin = *p1;
  rect->pmax = *p2;
}

int main(void)
{
  struct Punkt_s punkt1, punkt2;
  struct Rechteck_s myRecht;

  punkt1.x = 10; punkt1.y = 20;
  punkt2.x = 15; punkt2.y = 25;

  neuesRechteck(&punkt1, &punkt2, &myRecht);
}
```

Vergleiche

- Vergleichsoperatoren (also auch ==) können **nicht** direkt auf Strukturen angewandt werden. Folgendes geht also **nicht**:

```
struct Punkt_s p1, p2;
…
if(p1 == p2)   // geht nicht
{
   …
}
```

- Vergleiche müssen daher elementweise erfolgen
  - am besten durch eine separate Funktion

Vergleiche (2) – Beispiel

Verwendung einer Funktion zum Vergleich von Strukturen

```
#define FALSE 0
#define TRUE 1
int PunkteGleich(struct Punkt_s *p1, struct Punkt_s *p2)
{
  int ret = FALSE;

  if(p1->x == p2->x && p1->y == p2->y)
    ret = TRUE;

  return ret;
}
```

- in dieser Funktion könnten p1, p2 verändert werden
- werden sie aber nicht
- wie kann man das dem Compiler mitteilen?

Aufruf z.B. mit

```
struct Punkt_s p1, p2;
…
if(PunkteGleich(&p1, &p2) == TRUE) …
```

Konstante Zeiger (1)

- Zwei verschiedene Arten:
  - Konstanter Speicherbereich
    - Speicher, auf den der Zeiger verweist, wird als konstant definiert
    - Schreibender Zugriff ist nicht möglich
  - Konstanter Verweis
    - Zeiger kann nicht modifiziert werden
    - Er weist konstant auf ein bestimmtes Objekt

Konstante Zeiger (2)

```
const char *ptr;  // Das Objekt, auf das ptr verweist,
                     ist konstant.

char * const ptr = &var;        // Die Adresse in ptr ist
                                   konstant.

const char * const ptr = &var;     // Sowohl das Objekt
                                      als auch der
                                      Zeiger sind
                                      konstant.
```

Vergleiche (2) – Beispiel

Verwendung einer Funktion zum Vergleich von Strukturen

```
#define FALSE 0
#define TRUE 1
int PunkteGleich(const struct Punkt_s * const p1,
                 const struct Punkt_s * const p2)
{
  int ret = FALSE;

  if(p1->x == p2->x && p1->y == p2->y)
    ret = TRUE;

  return ret;
}
```

Aufruf z.B. mit

```
struct Punkt_s p1, p2;
…
if(PunkteGleich(&p1, &p2) == TRUE) …
```

const-Verwendung

- sollte man const so verwenden?
- ja → so restriktiv wie möglich

- warum? – wird ja alles nur komplizierter …
  - Programmiersicherheit
    - Compiler wird eine Fehlermeldung ausgeben, wenn man die Variable versehentlich doch verändern möchte
    - Wirkung nach außen ist ausgeschlossen – verbindet Vorteile von call-by-value und Verwenden von Zeigern
  - automatische Optimierung
    - Compiler optimiert Code beim Übersetzen automatisch
      - Zeit oder Speicher
    - geht besser, wenn er weiß, dass Variablen sicher nicht verändert werden

Zusammenfassung

- Zeiger enthalten Adressen
- Adressoperator &
- Inhaltsoperator *
- Verwendung
  - Veränderung von Variablen in Funktionen mit Außenwirkung
  - Rückgabe von mehreren Werten aus Funktionen
  - Vermeidung des Kopierens großer Datenmengen

- struct
  - im Prinzip wie elementare Datentypen
  - vereinfachte Schreibweise für Elementzugriff ->