Praxis der Programmierung

Ausblick: Was es in C noch gibt.

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Weitere Operatoren

Der Bedingungsoperator

`A ? B : C`

1. Auswertung von `A`
2. Wenn Wert von `A` ungleich `0`, Auswertung von `B` (= Rückgabewert);
3. sonst Auswertung von `C` (= Rückgabewert)
4. Rückgabetyp ist der „höhere“ Typ von `B` oder `C`

entspricht

```
if (A) return B;
else return C;
```

Besipiel: `(3>4) ? 5.0 : 6  // gibt 6.0 zurueck`

Bitoperatoren

UND-Operator	&
ODER-Operator	|
Exklusives ODER	^
Negationsoperator	~
Rechtsshift-Operator	>>
Linksshift-Operator	<<

- Die logischen Operatoren beziehen sich immer auf alle Bits ihrer Operanden.
- Sind anwendbar auf ganzzahlige Operatoren.
- Vorsicht bei Vorzeichen-behafteten Operanden.

Logische Bit-Operatoren

```
0 & 1        //  0 & 1 = 0
14 & 1       //  1110 & 0001 = 0000
```

Logische Bit-Operatoren

```
0 & 1        //  0 & 1 = 0
14 & 1       //  1110 & 0001 = 0000

0 | 1        //  0 | 1 = 1
14 | 1       //  1110 | 0001 = 1111
```

Logische Bit-Operatoren

```
0 & 1        //  0 & 1 = 0
14 & 1       //  1110 & 0001 = 0000

0 | 1        //  0 | 1 = 1
14 | 1       //  1110 | 0001 = 1111

0 ^ 1        //  0 ^ 1 = 1
14 ^ 1       //  1110 ^ 0001 = 1111 / Invertieren
14 ^ 3       //  1110 ^ 0011 = 1101 / von Bits!
```

Logische Bit-Operatoren

```
0 & 1        //  0 & 1 = 0
14 & 1       //  1110 & 0001 = 0000

0 | 1        //  0 | 1 = 1
14 | 1       //  1110 | 0001 = 1111

0 ^ 1        //  0 ^ 1 = 1
14 ^ 1       //  1110 ^ 0001 = 1111 / Invertieren
14 ^ 3       //  1110 ^ 0011 = 1101 / von Bits!

unsigned char a, b;
a = 9;       //  a = 0000 1001
b = ~a;      //  b = 1111 0110
```

Shift-Operatoren

`A >> B`

Verschieben der Bits von A um B Bitstellen nach rechts;
Auffüllen der vorderen Bits mit Nullen (Division durch 2^B)

```
unsigned char a;
a = 8;      //  0000 1000
a = a >> 3; //  0000 0001
```

Shift-Operatoren

`A >> B`

Verschieben der Bits von A um B Bitstellen nach rechts;
Auffüllen der vorderen Bits mit Nullen (Division durch 2^B)

```
unsigned char a;
a = 8;      //  0000 1000
a = a >> 3; //  0000 0001
```

`A << B`

Verschieben der Bits von A um B Bitstellen nach links;
Auffüllen der nachrückenden Bits mit Nullen (Multiplikation mit 2^B)

```
unsigned char a;
a = 8;      //  0000 1000
a = a << 3; //  0100 0000
```

Unionen und Bitfelder

Unionen

- **Deklaration:** wie Strukturen, mit `union` statt `struct`
- Members (Komponenten) stellen Alternativen dar;
  zu jedem Zeitpunkt ist nur eine Komponente gespeichert
- **Zugriffe** mit Punkt- bzw. Pfeilschreibweise
- **Beispiel:** Repräsentation eines Punktes entweder
  mit karthesischen oder Radial-/Polarkoordinaten

Unionen: Beispiel

```
struct cartPoint {
    double x, y;
}

struct radPoint {
    double radius, phi
}

struct point {
    enum pointType ptype;
    union coordinates c;
}
```

```
union coordinates {
    struct cartPoint cpt;
    struct radPoint rpt;
}

enum pointType {
    CART, RAD
}
```

Bitfelder

- bestehen aus vorgegebener Anzahl von Bits
- Bitmuster werden als Werte des Ganzzahltyps interpretiert
- **Deklaration:** `typ name:bitanzahl;`

```
unsigned a:4;   //  Werte von 0 bis 15
a = 3;          //  0011
a = 19;         //  0011
signed b:4;     //  Werte -8 bis +7
```

- plattformabhängige Interpretation
- für hardwarenahe und Graphikprogrammierung eingesetzt

Präprozessoranweisungen

Präprozessoranweisungen (1)

- Ausführung vor der eigentlichen Kompilation
- `#Direktive Text`
- *bisher:*
  - Einfügen von Dateiinhalten mit `#include datei`

    ```
    #include <stdio.h>
    ```

  - Vereinbarung symbolischer Konstanten und Makros mit

    ```
    #define Bezeichner Ersatztext
    #define double_inc(a,b) a++; b++;
    ```

Präprozessoranweisungen (2)

- Makros mit Parametern
  - `#define Bezeichner(Parameter1, ..., ParameterN) Ersatztext`
  - formale Parameter ohne Typ (flexibel, aber unsicher!)
  - Einsetzung des Ersatztextes mit aktuellen Parametern für die formalen Parameter

```
#define sum(n) n*(n+1)/2
...
int a = 12;
int s = sum(a);
...
```

- Aufheben von Makrovereinbarungen mit `#undef Bezeichner`

Präprozessoranweisungen (3)

- bedingte Kompilierung
  - Präprozessor entscheidet anhand von Ausdrücken und Symbolen, welche Codeteile übersetzt werden sollen
  - Direktiven `#if #elif #else ...`
  - Einsatz z.B. um auf Compilerversionen reagieren zu können oder Programmversionen in einer Datei zu halten (*Write once!*)
- eigene Fehlermeldungen während des Kompilierens generieren
- ...

Die Standardbibliothek

Bibliotheksfunktionen (1)

- **Funktionen zur Ein- und Ausgabe (stdio.h)**
  - Schreiben von Zeichen oder Strings auf `stdout`
  - Lesen von Zeichen oder Strings von `stdin`
  - Lesen oder Schreiben von bzw. in Streams/Dateien, ...
- **String- und Speicherbearbeitung (string.h)**
  - Vergleich, Kopieren zweier Strings
  - Suchen von Zeichen oder Teilstrings in Strings
  - Bestimmen der Länge eines Strings, ...
  - analoge Funktionen für Datenblöcke im Speicher

    z.B. `strcpy()` *vs.* `memcpy()`, ...

Bibliotheksfunktionen (2)

- **Mathematische Funktionen (math.h)**

  `sin(), cos(), log(), log10(), floor(), pow(), ...`

- **Zahlenkonvertierungen, Speicherverwaltung, Zufallszahlen (stdlib.h)**

  – `abs(), atof(), atoi(), atol(), ...`
  – `malloc(), realloc(), free(), ...`
  – `rand(), ...`

- **Klassifizierung und Konvertierung von Zeichen (ctype.h)**

  – Test ob ein Character alphanumerisch, ein Buchstabe, eine Zahl, ... ist
  – `tolower(), toupper()`

Bibliotheksfunktionen (3)

- **Festlegung länderspezifischer Zeichen und Darstellungen (locale.h)**
- **Datum und Uhrzeit (time.h)**
- einige mehr, versionsabhängig (s. Compiler-Handbuch)

Speicherklassen

Konzept der Speicherklassen

- Modifikation der Gültigkeit von Variablenvereinbarungen mit Hilfe von Schlüsselwörtern `auto`, `extern`, `static`, `register`
- ergänzen die Unterscheidung von lokalen und globalen Variablen

  lokal: Definition innerhalb eines Funktionsblocks

  – Verwaltung in den Stackframes zu Aufrufen der Funktionen
  – Verwendbarkeit nur, solange der Stackframe exisitiert

  global: Definition außerhalb aller Funktionsblöcke

  – gültig während des gesamten Programmlaufs
  – können von allen Funktionen verwendet werden

Speicherklassen `auto` und `extern`

- **auto**
  - Default-Speicherklasse
  - Verwendung, um auf Gültigkeitsbereich im Quellcode expilizit hinzuweisen

Speicherklassen `auto` und `extern`

- **auto**
  - Default-Speicherklasse
  - Verwendung, um auf Gültigkeitsbereich im Quellcode expilizit hinzuweisen
- **extern**
  - bewirkt reines *Deklarieren*:
    Name wird bekannt gegeben, aber noch kein Speicherplatz reserviert
  - Definition kann an anderer Stelle, sogar in anderer Datei (z.B. Header-Datei) erfolgen
  - gebräuchlich für globale Variablen

Speicherklassen `register` und `static`

- **register**
  - zur Definition spezieller `auto`-Variablen
  - „Bitte“ an Laufzeitsystem, schnellen Zugriff durch Anlegen in *Registern* des Prozessors zu ermöglichen (z.B. bei sehr häufigem Zugriff)
  - bevorzugte Behandlung kann nicht garantiert werden

Speicherklassen `register` und `static`

- **register**
  - zur Definition spezieller `auto`-Variablen
  - „Bitte“ an Laufzeitsystem, schnellen Zugriff durch Anlegen in *Registern* des Prozessors zu ermöglichen (z.B. bei sehr häufigem Zugriff)
  - bevorzugte Behandlung kann nicht garantiert werden

- **static**
  - *statische* lokale Variablen bis zum Ende des Programmlaufs gültig
  - ist aber nur in dem Block sichtbar/verwendbar, in dem sie definiert wurde
  - geeignet für Funktionen, die Informationen aus dem letzten Aufruf für den nächsten Aufruf bereitstellen wollen

Beispiel `static`

```
int globvar = 1;

void func() {
    static int statvar = 1,
    int locvar = 1,
    // alle 3 Variablen ausgeben
    statvar++;
    globvar++;
    locvar++;
}
```

```
int main() {
    func(); func(); func();
    // locvar  = 8; -> Compilerfehler!!!
    // statvar = 8; -> Compilerfehler!!!
    globvar = 8;   // o.k.
    func(); func();
}
```

Beispiel `static`

```
int globvar = 1;

void func() {
    static int statvar = 1,
    int locvar = 1,
    // alle 3 Variablen ausgeben
    statvar++;
    globvar++;
    locvar++;
}
```

```
int main() {
    func(); func(); func();
    // locvar  = 8; -> Compilerfehler!!!
    // statvar = 8; -> Compilerfehler!!!
    globvar = 8;   // o.k.
    func(); func();
}
```

```
statisch: 1  global: 1  lokal: 1
statisch: 2  global: 2  lokal: 1
statisch: 3  global: 3  lokal: 1
statisch: 4  global: 8  lokal: 1
statisch: 5  global: 9  lokal: 1
```