Programmiertechnik 1

Unit 6: Programmiersprache C – integrale Datentypen

Ablauf

- Datentypen
- Operatoren
- Ausdrücke
- Typkonvertierungen
- Vorrangregeln

Datenstruktur

- Kombination aus Datentyp und Operationen nennt man Datenstruktur
  - Häufig synonyme Benutzung der Begriffe
  - Mitunter Formulierung von Vor- und Nachbedingungen
    - Eiffel, CUnit-Assertions, keine native Unterstützung in C
- Variablen und Konstanten
  - Grundlegende Datenobjekte in einem Programm
  - Deklarationen enthalten Variablennamen und Typen
  - Alle Variablen müssen vor Benutzung deklariert werden
  - Initialisierung
- Ausdrücke berechnen Werte
  - Auch zur Initialisierung verwendbar (berechnet zur Compile-Zeit)
  - Konstante Ausdrücke
  - const Variablen – können sich zur Laufzeit nicht ändern
- Skalare (Integrale) vs. strukturierte Typen

Variablen

- Namen für Variablen
  - Bestehen aus Buchstaben und Zahlen
  - Erstes Zeichen muss Buchstabe sein, „_“ ist ein Buchstabe
    - Programm sollte keine Variablen enthalten die mit _ beginnen
    - Namenskonflikte mit Bibliotheken denkbar
  - 31 signifikante Zeichen (mindestens)
    - Groß- / Kleinschreibung werden unterschieden
    - Funktionen und externe Variablen haben womöglich weniger signifikante Zeichen (abhängig von Assembler, Linker)
    - Standard garantiert hier 6 signifikante Zeichen und _keine_ Gros-/Klein-Unterscheidung
  - Schlüsselworte sind reserviert, können kein Variablennamen sein
  - Konvention:
    - Kurze Namen für lokale Variablen, Loop-Indizes, etc.
    - Lange, beschreibende Namen für externe Variablen

Datentypen und Größen

- Nur wenige skalare (basic) Datentypen in C:
  - char - ein einzelnes Byte, ein Buchstabe im lokalen Zeichensatz (8 bit)
  - int - eine ganze Zahl, Größe entspricht Maschinenwort (16/32/64 bit)
  - float - Gleitkommazahl mit einfacher Genauigkeit (32 bit)
  - double - Gleitkommazahl mit doppelter Genauigkeit (64 bit)
- Qualifizierer:
  - long, short – anwendbar für Integers; Typ int darf dann weggelassen werden
    - long int a; short int b; entspricht:
    - long a; short b;
    - Häufig: short 16 bit, long 32/64 bit, int 16/32/64 bit
    - sizeof(short) <= sizeof(int) <= sizeof(long)
  - signed, unsigned
    - 2er-Komplementzahlen vs. Binärzahlen (ohne Vorzeichen)
    - Druckbare Zeichen sind immer positiv – char kann vorzeichenbehaftet sein

Größen ermitteln

- Manche Datenrepräsentationen sind maschinenabhängig
  - long double – Gleitkommazahlen mit erweiterter Genauigkeit
  - float, double – enstsprechend IEEE 754
  - <limits.h> <float.h> konsultieren
- Beispiel:
  - sizes.c ...

```
printf("Size of Char %d, sizeof returns %d\n",CHAR_BIT, sizeof( char ));
printf("Size of Char Max %d\n", CHAR_MAX);
printf("Size of Char Min %d\n", CHAR_MIN);
```

- Definitionen (Header-Files) residieren in
  - /usr/include
  - /usr/include/sys

sizes.c

Konstanten

- Integer-Konstanten
  - 1234 – int
  - 123456789L (oder l) – long;
    - Compiler konvertiert Werte, die nicht in int passen automatisch nach long
  - Suffix U (oder u) – unsigned
  - Werte:
    - Dezimal, Oktal, Hexadezimal:
    - Führende 0: 034 – Oktaldarstellung
    - Führendes 0x: 0xFF – Hexadezimaldarstellung
    - Bsp: 31 == 037 == 0x1f == 0X1F
    - Suffixes UL möglich: unsigned long n = 0XFUL; /* Wert ist 15 */
- Gleitkomma-Konstanten
  - 123.4 – mit Dezimalpunkt
  - (1e-2) – mit Exponent – oder beidem (123.4e-3)

Zeichen-Konstanten

- Integer-Konstante, deren Wert das Zeichen x im Zeichensatz angibt
  - Geschrieben: char a = ‘x‘;
  - Wert → Position von x im Zeichensatz (ASCII, EBCDIC, UTF-8)
    - Bsp: ‘0‘ == 48 im ASCII-Satz
    - Wert einer Ziffer x: int val = x - ‘0‘; /* besser als x-48 – portabel */
  - Escape-Sequenzen
    - ‘\n‘ – newline, ein-Byte-Zeichen, keine Folge!
    - Beliebige Zeichen: ‘\ooo‘ – 1-3 Oktalziffen ; ‘\xhh‘ – 1-2 Hexziffern
    - Bsp: # define VTAB ‘\013‘ /* ASCII, vertikaler Tabulator */
      # define BELL ‘\007‘ /* ASCII, Bell Zeichen */

Zeichen	Wert	Zeichen	Wert
\a	alert (bell) character	\\	backslash
\b	backspace	\?	Fragezeichen
\f	formfeed	\‘	einfaches Anführungszeichen
\n	newline	\“	doppeltes Anführungszeichen
\r	carriage return	\ooo	Oktalzahl
\t	horizontal tab	\xhh	Hexadzimalzahl
\v	vertical tab		

Konstante Ausdrücke

- Ausdruck aus Konstanten
  - Kann zur Übersetzungszeit berechnet werden
  - Kann überall dort auftreten, wo Konstanten stehen dürfen

```
# define MAXLINE 1000
char line[MAXLINE+1];

# define LEAP                /* in Schaltjahren */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

- Zeichenkettenkonstante:
  - Folge aus 0 oder mehr Zeichen in doppelten Anführungsstrichen

```
char * s1 = “I am a string“   ““;   /* leere Zeichenkette */
char * s2 = “Hello,“ “world“ ;     /* Verkettung zur Compile-Zeit */
```

  - String constant ist ein Feld von Zeichen; interne Repräsentation fügt 0-byte an

Zeichenketten (strings)

- Wieviele Bytes benötigt eine Zeichenkette?

```
/* gibt laenge von s zurueck */
int strlen( const char s[] ) {
    int i = 0;
    while (s[i] != '\0') i++;
    return i;
}
```

- Zeichenketten können beliebig lang sein
  - Speicherplatz ist Länge+1 – für abschließendes Nullbyte
  - Länge kann nur durch Ablaufen der gesamten Kette ermittelt werden
- Funktionen zur Behandlung von Zeichenketten
  - strlen(), strcmp(), etc – deklariert in <string.h>
  - Falle: if (s1 == s2)... - hier werden nur zwei Zeiger verglichen
  - Richtig: if (strcmp(s1, s2) == 0)...
- Unterschied: Zeichenketten-Konstante vs. Zeichen-Konstante
  - “a“ /* 2 byte */ vs. ‘a‘ /* 1 byte */

strlen.c

Aufzählungskonstanten

- Enumeration constant
- Liste konstanter Integer-Werte
  - enum boolean { NO, YES };
    - NO == 0; YES == 1;
    - Werte vom Compiler generiert
  - Namen müssen verschieden sein, Werte können übereinstimmen
    - enum escapes { BELL = '\a', BACKSPACE = '\b', TAB = '\t', NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };

- Enumerations sind besser als #define-Konstanten
  - Debugger kann symbolische Werte anzeigen
  - Werte können vom Compiler generiert werden
    - Aber: Compiler überprüft Werte bei Zuweisungen nicht
    - Man kann also auch Werte ausserhalb des aufgezählten Bereiches an Variablen vom Aufzählungstyp zuweisen (Falle!)

Deklarationen

- Variablen müssen vor Benutzung deklariert werden
  - Deklaration gibt Type und mehrere Variablen an
    - int lower, upper, step;
    - char c, line [1000];
  - Variablen können über mehrere Deklarationen verteilt sein
    - int lower;
    - int upper;
    - int step;
  - Variablen können initialisiert werden
    - char esc = '\\';
    - int i = 0;
    - int limit = MAXLINE + 1;
    - float eps = 1.0e-5;
  - Initialisierung erfolgt einmal, vor Programmausführung
  - Ausnahme: lokale (automatic) Variablen
  - Externe und statische Variablen werden auf 0 initialisiert (lokale nicht...)

Deklarationen (contd.)

- Explizit initialisierte Variablen erhalten ihren Wert bei jedem Eintritt in den umgebenden Block

```
int exp( int g ) {
    int i = 1;
    while (g-- > 0) i = i*2;
    return i;
}
```

- Variablen können als const markiert werden

```
const double e = 2.71828182845905;
const char msg[] = ”warning: ”;
```

- Auch Funktionsargumente können als Konstanten markiert sein

```
int strlen( const char[] );
```

- Zuweisungen an Konstanten werden vom Compiler als Fehler ausgewiesen (implementierungsabhängig)

exp.c

Arithmetische Operatoren

- Binäre arithmetische Operatoren: +, -, *, /, modulo (%)
- Division schneidet gebrochenen Rest ab

  x%y liefert Rest

- Beispiel:

```
if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0))
   printf("%d ist ein Schaltjahr\n", year);
else
   printf("%d ist kein Schaltjahr\n", year);
```

- % kann nicht auf float oder double angewandt werden
- Abschneiden bei / und Vorzeichen bei % sind maschinenabhängig für negative Zahlen; ebenso Verhalten bei Überlauf und Unterlauf
- Vorrang: (+, -) < (*,/,%) < (einstelliges +, -)
- Assoziativität (Auswertungsreihenfolge): von links nach rechts

Relationale/logische Operatoren

- Vergleichsoperatoren >, >=, <, <= alle mit gleichem Vorrang
- Gleichheitsoperatoren ==, != haben niedrigeren Vorrang
- Relationale Operatoren haben niedrigeren Vorrang als arithmetische

  i < lim-1 == i < (lim-1)
- Logische Operatoren sind interessant: && (und) || (oder)
  - Auswertung von links nach rechts
  - Auswertung stoppt, sobald Wert des Ausdrucks entscheidbar
  - Anwendung in C-Programm:

```
for (i=0; i < sizeof(s)-1 && (c=getchar()) != '\n' && c != EOF; i++)
  s[i] = c;
s[i+1] = 0;
```

  - Wir müssen _vor_ dem Lesen testen, ob noch Platz im Feld ist
- Einstellige Operatoren: if (valid == 0)... Ist äquivalent zu: if (!valid)...

getline.c

Typkonvertierungen

- Operanden verschiedener Typen werden konvertiert
  - Automatisch: „schmalerer“ zu „weiterem“ Typ:

```
int i;
float f, g;
g = f + i;       /* i wird automatisch zu float konvertiert */
```

  - Automatisch, mit Compiler-Warnung:

```
i = f + g;       /* Warnung, potentieller Informationsverlust */
```

- char ist Integer-Type
  - Kann in Ausdrücken frei mit Integern gemischt werden

```
int atoi( const char s[] ) {                /* atoi: convert s to integer      */
        int i, n = 0;
        for (i=0; s[i] >= ´0´&& s[i] <= ´9´; i++) n = 10*n + (s[i]-´0´);
        return n;
}
```

  - Mehr: isdigit(), tolower(), toupper()
  - <stdlib.h> und <ctype.h> enthalten derartige Konvertierungsroutinen

Kann char negativ sein?
Nicht für druckbare Zeichen!
Sonst: unsigned char verwenden!

Konvertierungen (contd.)

- Relationale und logische Ausdrücke liefern 1 (true) oder 0 (false)
  → können in Integer-Ausdrücken verwendet werden
  d = c >= ´0´ && c <= ´9
  → d wird 1 wenn c eine Ziffer ist, sonst 0
- Implizite arithmetische Konvertierungen
  - Niedriger Type wird „promoted“, dann wird Operation ausgeführt
  - Regeln:
    - Ein Operand long double → anderen Operanden nach long double konvertieren
    - Sonst, Operand double → anderen Operanden nach double konvertieren
    - Sonst, Operand float → anderen Operanden nach float konvertieren
    - Sonst, Operand char oder short → nach int konvertieren
    - Dann, ein Operand long → anderen Operanden nach long konvertieren
  - float wird nicht automatisch nach double konvertiert
- Explizite Typecasts verwenden! (<typename>) expression

Inkrement / Dekrement

- Zwei ungewöhnliche Operatoren:
  - Operator ++ inkrementiert Operanden um 1
  - Operator -- dekrementiert Operanden um 1
- Interessant:
  - ++ und -- können als prefix oder postfix Operatoren verwendet werden
  - ++n oder n++ : in beiden Fällen wird n inkrementiert
    - ++n erhöht n bevor der Wert von n verwendet wird
    - n++ erhöht n nachdem der Wert von n verwendet wurde

```
n = 5;
x = n++; /* x == 5, n == 6 */
x = ++n; /* x == 7, n == 7 */
```

  - Inkrement / Dekrement können nur auf Variablen angewendet werden
    - (i+j) ++; ist illegal

Beispiel ++

- All Vorkommen von c sollen aus Zeichenkette s gelöscht werden
- Jedes nicht-c wird in die aktuelle j-Position kopiert

```
/* squeeze: delete all c from s */
void squeeze(char s[], int c) {
        int i, j;

        for ( i = j = 0; s[i] != '\0'; i++ )
                if ( s[i] != c )
                        s[j++] = s[i];          /* entspricht s[j] = s[i]; j++; */
        s[j] = '\0';
}
```

- Kompaktere Schreibweise
- Weniger Hauptspeicherzugriffe
- Vorsicht bei Funktionsaufrufen

  f(n, n, n++) – Reihenfolge der Argumentauswertung nicht definiert

squeeze.c

Bitweise Operatoren

- Bit-Manipulation
  - nur auf char, short, int, long anwendbar

&	Bitweises UND
|	Bitweise inklusives ODER
^	Bitweises exklusives ODER
<<	Verschieben aller Bits nach links
>>	Verschieben aller Bits nach rechts
~	Einer-Komplement (Invertieren aller Bits)

  - Bits löschen: n = n & <bitmask>
  - Bits setzen: n = n | <bitmask>
  - Bitweises exklusives ODER setzt
    - 1 wenn Operanden sich in einer Bitposition unterscheiden
    - 0 wenn Operanden in einer Bitposition übereinstimmen

Nicht mit && oder || verwechseln!

Shift-Operatoren

- x << n oder x >> n
  - Operand wird um n Stellen verschoben
  - n muss positiv sein
  - Entspricht Multiplikation (<<) oder Division (>>) mit 2^n
- Left shift:
  - Freie bits werden mit Null aufgefüllt
- Right shift:
  - Unsigned: Freie bits werden mit Null aufgefüllt
  - Signed: maschinenabhängig
    - „arithmetic shift“ – Auffüllen mit Vorzeichen-Bit
    - „logical shift“ – Auffüllen mit 0-bits

bit_invert.c

Zuweisungsoperator

- Assignment operator:
  - i += 2 ist äquivalent zu i = i + 2
- Viele binäre Operatoren haben entsprechenden Zuweisungsoperator:
  - op $\in$ { +, -, *, /, %, <<, >>, &, ^, | }
  - Sind $expr_1$ und $expr_2$ Ausdrücke, so gilt:
    - $expr_1$ op= $expr_2$ ist äquivalent zu
    - $expr_1$ = ($expr_1$) op ($expr_2$)
  - $expr_1$ wird nur einmal ausgewertet (Seiteneffekte!!)
- Klammersetzung:
  - x *= y + 1; bedeutet x = x * (y + 1); /* nicht x = x * y + 1; */
- Effizienz, Lesbarkeit
- Zuweisungsoperator hat selbst einen Wert (nämlich den zugewiesenen)
  - Kaskadierung: while ((c = getchar()) != EOF) ...

Bedingte Ausdrücke

- Dreistelliger Operator ?:
  - $expr_1$? $expr_2$: $expr_3$
  - $expr_1$ wird ausgewertet
    - → wenn Wert != 0 (true): $expr_2$ wird ausgewertet; ergibt Wert des bedingten Ausdrucks
    - → Sonst: $expr_3$ wird ausgewertet; ergibt Wert des bedingten Ausdrucks
- Bsp: Maximum berechnen
  - if (a > b) z = a; else z = b;
  - z = (a > b) ? a : b;
- ?: ist ein Ausdruck (expression)
  - Kann in Berechnungen verwendet werden
  - Unterliegt Typkonvertierungen
    - int n; float f;
    - Welchen Typ hat (n > 0) ? F : n

Typ ist float; unabhängig ob n positiv

lower.c

Vorrang und Auswertungsreihenfolge

Operatoren	Assoziativität (Auswertungsreihenfolge)
() *(function)* [] *(array)* -> *(member)* .	Von links nach rechts
! ~ ++ -- + - * & (type) sizeof	Von rechts nach links
* / %	Von links nach rechts
+ -	Von links nach rechts
<< >>	Von links nach rechts
< <= > >=	Von links nach rechts
== !=	Von links nach rechts
& *(bitweises UND)*	Von links nach rechts
^ *(bitweises exklusives ODER)*	Von links nach rechts
| *(bitweises ODER)*	Von links nach rechts
&&	Von links nach rechts
||	Von links nach rechts
? :	Von rechts nach links
= += -= *= /= %= &= ^= |= <<= >>=	Von rechts nach links
,	Von links nach rechts

Einstellige Operatoren (Vorzeichen, Zeiger-Dereferenzierung)

+, -, *

haben Vorrang vor zweistelligen Formen

Auswertungsreihenfolge (contd.)

- C legt Auswertungsreihenfolge von Operanden nicht fest
  - Ausnahmen: &&, ||, ?:, ’,’
  - Bsp: x = f() + g(); -- f könnte vor g gerufen werden (oder umgekehrt)
- Falle:
  - Falsch: printf(”%d, %d\n“, ++n, power(2, n));
    - Ausgabe kann sein: n+1, 2^{n+1}
    - Oder: n+1, 2^{n}
  - Richtig: ++n; printf(”%d, %d\n“, n, power(2, n));
- Seiteneffekte:
  - Verursacht durch Funktionsaufrufe, verschachtelte Zuweisungen, Inkrement- und Dekrement-Operatoren
  - Eine Variable wird im Zuge der Auswertung eines Ausdrucks verändert
    - a[i] = i++; Subskript für a kann alter oder neuer Wert von i sein
    - Compiler-abhängig
- Code, der von der Auswertungsreihenfolge der Operanden abhängt, zeugt von schlechtem Programmierstil – in jeder Programmiersprache

Zusammenfassung

- Datentypen
  - Wenige integrale (skalare) Datentypen
  - Char is wie Integer; 2(3) Gleitkommatypen
  - Aufzählungstypen legen symbolische Namen für Integer-Werte fest
- Operatoren
  - Arithmetische und logische Operationen für integrale Datentypen
  - Bitweise Operationen
- Ausdrücke
  - Können verschachtelt sein
  - Auch die Zuweisung hat einen Wert, der weiter verarbeitet werden kann
- Typkonvertierungen
  - Automatisch: vom schmaleren zum weiteren Typ
- Vorrangregeln
  - Im Zweifel Klammern setzen (!)