Einführung in die Programmiersprachen C und C++

Prof. Dr. Ulf Rehmann, Fakultät für Mathematik

Übungsblatt 2 (8 Seiten)

`http://www.mathematik.uni-bielefeld.de/~rehmann/CC++/`

Elementare Datentypen in C und C++

- `char` – 1 Byte, kann 'einen Charakter' (z. B. `'a', '3', '&', ...`) enthalten.
- `int` – ganze Zahl, Größe ist implementationsabhängig.
- `float` – Gleitkommazahl einfacher Genauigkeit.
- `double` – Gleitkommazahl doppelter Genauigkeit.

`int` kann qualifiziert werden durch `unsigned, short, long` zu `unsigned int, short int, long int`, oder kurz zu `unsigned, short, long`.
`char` kann qualifiziert werden zu `unsigned char`

Bereichsbeispiele:

	PC		Viele Workstations	
`char`	`-128 -- +127`	(1 Byte)	`-128 -- +127`	(1 Byte)
`short`	`-32768 -- +32767`	(2 Bytes)	`-32768 -- +32767`	(2 Bytes)
`int`	`-32768 -- +32767`	(2 Bytes)	(4 Bytes)	
`long`	`-2147483648 -- +2147483647`	(4 Bytes)	(8 Bytes)	

Die Fließkommatypen sind gleich für PC/Workstations (IEEE-Format) (siehe `/usr/include/float.h`):

`float`	$\pm$	1.710^{-38}	—	3.410^{+38}	(4 Bytes)
`double`	$\pm$	2.210^{-308}	—	1.710^{+308}	(8 Bytes)

Schlüsselwörter von C und C++ – und damit verboten als Namen – sind die folgenden (mit $^{++}$ sind Schlüsselwörter von C++ bezeichnet, die nicht auch Schlüsselwörter von C sind):

`and`$^{++}$	`const`	`false`$^{++}$	`not`$^{++}$	`signed`	`typeid`$^{++}$
`and_eq`$^{++}$	`const_cast`$^{++}$	`float`	`not_eq`	`sizeof`	`typename`$^{++}$
`asm`	`continue`	`for`	`operator`$^{++}$	`static`	`union`
`auto`	`default`	`friend`$^{++}$	`or`$^{++}$	`static_cast`$^{++}$	`unsigned`
`bitand`$^{++}$	`delete`$^{++}$	`goto`	`or_eq`	`struct`	`using`$^{++}$
`bitor`$^{++}$	`do`	`if`	`private`$^{++}$	`switch`	`virtual`$^{++}$
`bool`$^{++}$	`double`	`inline`$^{++}$	`protected`$^{++}$	`template`$^{++}$	`void`
`break`	`dynamic_cast`$^{++}$	`int`	`public`$^{++}$	`this`$^{++}$	`volatile`
`case`	`else`	`long`	`register`	`throw`$^{++}$	`wchar_t`$^{++}$
`catch`$^{++}$	`enum`	`mutable`$^{++}$	`reinterpret_cast`$^{++}$	`true`$^{++}$	`while`
`char`	`explicit`$^{++}$	`namespace`$^{++}$	`return`	`try`$^{++}$	`xor`$^{++}$
`class`$^{++}$	`export`$^{++}$	`new`$^{++}$	`short`	`typedef`	`xor_eq`$^{++}$
`compl`$^{++}$	`extern`				

C - und C++ - Operatoren:

1. Unäre Operatoren:

`+, -` (Vorzeichen), `!` (Negation), `~` (Bit-Komplementierung), `++` (Inkrement), `--` (Dekrement, beide als Prä- oder Postoperatoren), `*` (Umleitung, Indirection), `&` (Adressoperator), `sizeof` (liefert Größe eines Datentyps in Byte).

Operatoren, angewandt auf Ausdrücke, ergeben wieder Ausdrücke, die Werte haben. Die In-/Dekrement-Operatoren unterscheiden sich in ihrer Prä- oder Postfix-Notation:

```
int a, b; a = 5; b = a++;
```

Wert von `a` wird nach `b` kopiert, danach(!) wird inkrementiert; liefert als Resultat: `b` hat den Wert 5, `a` hat den Wert 6.

Dagegen:

```
int a, b; a = 5; b = ++a;
```

Wert von `a` wird erst(!) inkrementiert, der neue Wert von `a` wird nach `b` kopiert. Resultat: `b` hat den Wert 6, `a` hat den Wert 6.

2. Binäre Operatoren:

2.1. Arithmetische Operatoren:
`+, -, *, /, %` (Rest bei ganzzahliger Division)

2.2. Relationale (logische) Operatoren:
`<` (kleiner), `<=` (kleiner-gleich), `>` (größer), `>=` (größer-gleich), `==` (gleich), `!=` (ungleich), `&&` (logisches 'UND'), `||` (logisches 'ODER'),

2.3. Bit-arithmetische Operatoren:
`<<, >>` (Bitshift-Operatoren).
Beispiele:
`a << 3` verschiebt die Bitfolge von `a` um 3 Positionen nach links bei Einfügen von Nullen: $a = 5$ hat z. B. die Bitcodierung 101. Also: `a << 3` liefert $101000 \cong 40$ bei "Standard"-Bitcodierung.
`a >> 2` verschiebt die Bitfolge von `a` um 2 Positionen nach rechts: $a = 50$ hat die Bitcodierung 110010, `a >> 2` ergibt $1100 \cong 12$. (Siehe das Programm `bitshift.c`.)
`&` (bitweise 'UND'): `00101000 & 00110010` liefert `00100000`,
`|` (bitweise 'ODER'): `00101000 | 00110010` liefert `00111010`,
`^` (bitweise ausschließliches 'ODER'): `00101000 ^ 00110010` liefert `00011010`.
2.4. Cast-Operator: `(Typ-Name) Ausdruck` macht `Ausdruck` zu einem Objekt vom Typ `Typ-Name`.
Beispiel: `int a; double e=2.718; a = (int)e;` liefert den Wert $a = 2$.

2.5. Funktions- und Array-Operatoren:
`(), []`, Beispiele: `int ggt(int a, int b); char s[20];`

2.6. Komponenten-Operatoren für Strukturen: `.` und `->`. In C++ gibt es darüber hinaus: `.*` und `->*`

3. Konditionaloperator (ternär):
Syntax: `ausdruck1 ? ausdruck2 : ausdruck3`
Liefert als Wert `ausdruck2`, falls `ausdruck1` ungleich `0`, und `ausdruck3` sonst (vergleiche if - else).
Beispiel:`int a = 3, b = 5, c = ( a > b ) ? a : b;` liefert für `c` den Wert `5`.

4. Zuweisungsoperatoren:
`=` , Beispiel: `a = 17;`
Eine Zuweisung liefert einen Wert, nämlich den der zugewiesenen Größe, also sind Mehrfachzuweisungen wie `a = b = u = 17;` möglich.
`+=` , Beispiel: `a += 2` ist logisch äquivalent mit `a = a + 2`, analog für jeden (bit-)arithmetischen binären Operator: `a /= b` ist äquivalent mit `a = a/b` usw.

5. Kommaoperator: `,` (gruppiert Ausdrücke als Trennungszeichen oder Separator).

Operatorenpräzedenz in C:

Operatoren	Assoziativität
`() [] -> .`	links
`! ~ ++ -- + - * & (type) sizeof`	rechts
`* / %`	links
`+ -`	links
`<< >>`	links
`< <= > >=`	links
`== !=`	links
`&`	links
`^`	links
`|`	links
`&&`	links
`||`	links
`?:`	rechts
`= += -= *= /= %= &= ^= |= <<= >>=`	rechts
`,`	links

Die unären `+, -, *, &` haben höhere Präzedenz als die binären Formen. Die Assoziativität regelt für den Fall, dass ein Operand zwischen zwei Operatoren gleicher Priorität steht, ob der linke oder der rechte dieser Operatoren als erster genommen wird, falls beide Möglichkeiten sinnvoll sind:
`a - b + c` bedeutet also:
`(a - b) + c`, nicht etwa:
`a - (b + c)`, während
`a = b += 3` die Bedeutung
`a = (b += 3)` hat, oder,
noch ausführlicher:
`b = b+3, a = b`.

Beispiel 2.1: Testen Sie z. B. die Wirkungsweise des `?:`-Operators (cf. Nr. 3 oben) und des `<<`-Operators (cf. 2.3 oben) mit folgendem Programm:

```
/* test.c */
#include <stdio.h>

main() {
        int a, b;
        while (1) {
          printf("Eingabe von a und b: ");
          scanf("%d %d", &a, &b);
          printf("Maximum von %d, %d ist: %d\n", a, b, (a>=b ? a : b));
          printf("%d << %d liefert:       %d\n", a, b, a<<b );
        }
}
```

Aufgabe 2.1: Was bedeutet `-a++` *? Gibt es einen Unterschied zu* `-++a` *? Macht* `(-a)++` *einen Sinn?*

Aufgabe 2.2: Testen Sie mit einem kleinen Programm alle oben angegebene Operatoren auf ihre Wirkungsweise, indem Sie Werte von Ausdrücken ausgeben, in denen die Operatoren auftauchen.

Arrays (oder Felder) in C

Mit der Deklaration `char flags[1000];` deklariert man ein Array von 1000 Variablen vom Typ `char`, die die Namen `flags[0], flags[1], ..., flags[999]` haben. Analoges gilt auch für andere Datentypen: `int primzahlen[200];` deklariert ein Array von 200 Variablen vom Typ `int` mit den Namen `primzahlen[0], ..., primzahlen[199]`, usw.
Man verwendet diese Variablennamen genau wie gewöhnliche Variablennamen; mit

```
primzahlen[0] = 2;
primzahlen[1] = 3;
primzahlen[2] = 5;
...
```

weist man ihnen Werte zu, mit `printf("%d", primzahlen[199])`; druckt man sie aus, mit

```
int i = 17;
int z =  primzahlen[i];
```

weist man der `int`-Variablen `z` den Wert von `primzahlen[17]` zu, usw.
Ein Beispiel für die Verwendung von Arrays liefert das folgende Programm:

```
/* primzahlen.c                                                    */
/* Das zweitaelteste Computerprogramm der Welt-: von anno -250.    */
/* Copyright: ERATOSTHENES, Philologe, Informatiker und Geograph   */

#include <stdio.h>

#define TRUE  1                 /* Vorabdeklaration einiger Werte */
#define FALSE 0
#define MAX   4000000
char flags[MAX];           /* Deklaration eines Feldes ("array") von .. */
                           /* MAX Variablen flags[0], ...flags[MAX - 1] */
                           /* vom Typ "character"; 1 byte               */
```

```
main() {
      int i,prime,k,count,size;

      printf("Bestimmung aller Primzahlen bis "); scanf("%d", &size);
      size = (size+1)/2-2; count = 1; /* Nur ungerade Zahlen werden gesiebt */

      for (i = 0; i <= size; i++)     /* Initialisiere das ...              */
          flags[i]=TRUE;              /* ... ganze Feld                     */

      for (i = 0; i <= size; i++) {
          if (flags[i]) {           /* Ist flag[i]=TRUE, so ist i+i+3 prim */
                prime = i+i+3; k=i+prime;
                while (k<=size) {                /* also streiche alle ... */
                    flags[k] = FALSE; k += prime; /* seine nichttrivialen  */
                }                                /* Vielfachen            */
                count++;                         /* und zaehle.           */
           }
       }
       printf("\n%d Primzahlen\n",count);
}
```

Aufgabe 2.3: Modifizieren Sie das Programm `primzahlen.c` *derart, dass die Primzahlen der Reihenfolge nach ausgegeben werden.*

Aufgabe 2.4: Das Programm `primzahlen.c` *profitiert davon, dass es die geraden Zahlen, abgesehen von der 2, gar nicht erst in Betracht zieht, weil dies ja sowieso keine Primzahlen sind. Auf diese Weise reicht ein* `flags`*-Array etwa der Größe 1000, um alle Primzahlen bis 2000 zu bestimmen. Modifizieren Sie das Programm derart, dass von vornherein auch die durch 3 teilbaren Zahlen nicht betrachtet werden. Wieweit gelangt man dann mit einem* `flags`*-Array der Größe 1000?*

Das folgende Programm zeigt, wie man ein Array von Strings (= Zeichenketten oder Wörter) definieren und verwenden kann.

```
/* tage.c */

#include <stdio.h>

char *tage[] = { "Sonntag", "Montag", "Dienstag",
                 "Mittwoch", "Donnerstag", "Freitag", "Sonnabend" };

main() {
  int i;
  for (i=0; i<7; i++) {
    printf("Der %d-te Wochentag ist %s\n", i, tage[i]);
  }
  printf(">>> %c\n", tage[3][6]);
}
```

Aufgabe 2.5: Schreiben Sie ein Programm, das folgendes leistet: Nach Eingabe dreier Zahlen, etwa `13, 12, 2002`*, erfolgt die Ausgabe:* `13. Dezember 2002`*, und analog für andere Daten des laufenden Jahres.*

Hinweis: Definieren Sie ein Array `char *monat[]` derart, dass z. B. `monat[10]` den Wert `"Oktober"` hat. Für Ehrgeizige: Erweitern Sie das Programm, so dass die Ausgabe lautet: `Freitag, 13. Dezember 2002` usw. (Andere Jahre?)

Zur Syntax und Semantik von printf und scanf:
Vergleiche die Manual-Einträge: `man printf, man scanf`, die Zeilen mit "**??**" sind Mini-Aufgaben.

Beispiele:

`printf("Backslash gefolgt von n bedeutet\n \"Newline\"")` druckt:

```
Backslash gefolgt von n bedeutet
 "Newline"
```

`printf("%d", a)` druckt den Inhalt der (`int`-) Variablen `a` dezimal,
`printf("%o", b)` druckt den Inhalt der (`int`-) Variablen `b` oktal,
?? `printf("%x", c)` tut was?
?? `printf("%10.6f", d)` hält `d` für eine `float`-Variable und druckt?
?? Ersetze `f` im letzten Beispiel durch `e, g`. Was geschieht?
`printf("%s", string)` hält `string` für eine Zeichenkette,
`printf("%c", charact)` deutet `charact` als Charakter.
?? Was bedeuten die Control-Strings `"%10s", "%-20s", "%-20.4s"` usw. ?

Generell bedeutet der Ausdruck

`printf(<string> [, <var_1>, ..., <var_n>]);`

folgendes: Der Control-String `<string>` wird Zeichen für Zeichen auf den Bildschrim geschickt, mit folgenden Ausnahmen: Ein `%`-Zeichen in `<string>` leitet ein Interpretationsfeld innerhalb dieses Strings ein, das das Ausgabeformat einer der Variablen oder Ausdrücke `<var_1>, ..., <var_n>` festlegt, dabei bezieht sich das i-te `%`-Zeichen in `<string>` auf `<var_i>`.
`%` gefolgt von `d,o,x` setzt voraus, dass die zugehörige Variable (oder der entsprechende Ausdruck) `<var>` vom Typ `int` oder `char` ist, und gibt den Zahlenwert dezimal, oktal oder hexadezimal aus. Steht hinter dem %Zeichen eine Zahl n wie in `%4x` oder `%-10d`, wird die Zahl in einem Feld der Breite $|n|$ ausgegeben, und zwar rechtsbündig bei positiver Zahl, linksbündig bei negativer Zahl. Vor `d,o,x` kann ein `l` stehen, dann muss `var` vom Typ `long int` sein.
`%` gefolgt von `f, lf` erwartet `<var>` vom Typ `float` oder `double`, eine Zahl nach `%` und vor `f` oder `lf` wie in `%7.4f` legt das Augabeformat fest (in diesem Fall: Feldbreite 7 und 4 Nach"komma"zahlen), Vorzeichenregel wie oben. Es gibt weitere Ausgabe-Anweisungen für "wissenschaftliche" Notation etc., siehe `man printf`.
Soll ein `%`- oder `\`-Zeichen ausgegeben werden, verwendet man `%%` bzw. `\\` im Control-String.
`%c` wie in `printf("%c",x)` erwartet die Variable `x` vom Typ `char`, `%s` wie in `printf("%s",z)` erwartet, dass `z` einen "String" bezeichnet.
Rückgabewert von `printf()` ist die Anzahl der geschriebenen Zeichen, ein `int` ≥ 0.

Ähnliche Regeln gelten für die Argumente der Input-Funktion `scanf`. Beispiel:
`scanf("%d", &a)` erwartet (z. B. von der Tastatur) eine dezimale `int`-Eingabe und weist diese als Wert der (`int`-) Variablen `a` zu.
(Als Argument erscheint nicht `a` selbst, sondern `&a`, die "Adresse" oder ein auf a weisender "Pointer".)

Der allgemeine Aufruf von scanf sieht so aus:

`scanf(<string> [, <&var_1>, ..., <&var_n>]);`

die optionalen Variablen sind **Adressen** (vorangestelltes `&`).
(Wiederholte) Aufrufe von `scanf` behandelt den Strom (die Folge) der Eingabezeichen so: Der Eingabe-Strom muss den Zeichen des Control-Strings `<string>` entsprechen, wobei Konversionsspezifikationen (beginnend mit `%` wie oben) erwarten, dass die Eingabe entsprechende Datenformate liefert, die dann in die zugehörigen Variablen eingelesen werden. White Space (beliebiger Länge, incl. Länge = 0) im Control-String entspricht White Space beliebiger Länge im Eingabe-Strom, andere Zeichen im Control-String müssen im Eingabe-Strom einzeln "gematcht" werden.
Rückgabewert von `scanf()` ist die Anzahl der erkannten und zugewiesenen Felder (ein `int` ≥ 0), man kann also über den Rückgabewert erkennen, ob die Eingabe inkorrekt war (wenn dieser verschieden ist von der Anzahl der mit Wert zu belegenden Variablen).

Beispiele zur Syntax von `while` und `for`:
Die folgenden Beispiele 2.2 - 2.4 sind programmiertechnisch keineswegs alle vorbildlich. Sie sollen hier nur anhand eines einfachen Beispiels die formale Äquivalenz des `for`-Konstrukts mit dem in der Vorlesung angegebenen erweiterten `while`-Konstrukt verdeutlichen. Urteilen Sie selbst über die Zweckmäßigkeit und Lesbarkeit der verschiedenen Varianten.

Beispiel 2.2:

```
#include <stdio.h>

main() {
      int a, b, c;
      printf("Eingabe a, b : ");
      scanf("%d %d", &a, &b);
      while (b != 0) {
          c = a % b; a = b; b = c;
      }
      printf("%s %d\n","ggt =", a);

}
```

Bemerkung: Das `while`-Konstrukt in diesem Programm kann mit Hilfe des Komma-Operators auch so geschrieben werden:

```
 while (b != 0)
    c = a % b, a = b, b = c;
```

Aufgabe 2.6: Was geschieht, wenn die `scanf`*-Zeile ersetzt wird durch folgendes?*

```
scanf ("%d %d ", &a, &b);
```

Versuchen Sie, anhand des Manual-Eintrags von `scanf` *oder der oben gegebenen Erläuterungen eine Erklärung zu geben. Hinweis: Es ist hilfreich, das Programm in eine* `while(1)`*-Schleife einzubinden, um damit einen längeren "Eingabestrom" von Zhalen zu untersuchen.*

Beispiel 2.3:

```
#include <stdio.h>
main() {
      int a, b, c;
      printf("Eingabe a, b : ");

      for( scanf ("%d %d", &a, &b); b != 0; ) {
            c = a % b; a = b; b = c;
      }
      printf("%s %d\n","ggt =", a);
}
```

Beispiel 2.4:

```
#include <stdio.h>
main() {
      int a, b, c;
      printf("Eingabe a, b : ");
      for(scanf ("%d %d", &a, &b); b!=0; c = a % b, a = b, b = c)
              ;                                  /* leeres for-Statement */
      printf("%s %d\n","ggt =", a);
}
```

Die folgenden Beispiele sollen die bisher diskutierten Programmiertechniken weiter demonstrieren und Gelegenheit zur Wiederholung bieten. 2.5, 2.6 sind Beispiele zum Funktionsbegriff, 2.7 – 2.8 sind Abwandlungen des Idioms

```
while( (c=getchar()) != EOF)
        putchar(c)
```

2.7 ist eine Kombination dieser Dinge.

Beispiel 2.5

```
#include <stdio.h>

main() {
      int i;
      for ( i = 0; i < 10; i++)
          printf("%d %6d %6d\n", i, power(2,i), fact(i));
}

int power(int base, int n) {      /* berechnet base hoch n   */
    int i, p = 1;                  /* Beim Deklarieren wird p initialisiert */
    for (i = n; i > 0; i--)
        p = p * base;              /* Kuerzer: p *= base;     */
    return p;
}

int fact(int y) {             /* berechnet 1*2* ... *(y-1)*y    */
    int ergebnis = 1;
    while (y != 0)                 /* Achtung! Falls y < 0 ....      */
          ergebnis *= y--;  /* kurz fuer: ergebnis = ergebnis * y; y--; */
    return ergebnis;
}
```

Aufgabe 2.7: In beiden vorangegangenen Funktionen sind keine Überprüfungen der Argumente vorgesehen. Korrigieren Sie das.

Beispiel 2.6:

Beachten Sie die folgende Regel:
Eine Funktion, die einen Rückgabewert vom Typ Nicht-`int` hat, muss wie im nächsten Beispiel vor dem Erstaufruf *deklariert* sein durch Angabe ihres "Protopyps" (siehe Beispiel 2.6). Der Compiler nimmt sonst beim Erstaufruf an, dass die Funktion `int` zurückliefert, und bemängelt dann später bei der Definition der Funktion ein ‘‘`type mismatch`’’.

```
/* Quadratwurzel ohne Errorausgabe fuer negative Radikanden       */

#define  abs(A) ( (A) < 0 ? -(A) : (A) )   /* "Funktion" per define */
#define  DELTA  1.0e-16

double wurzel(double);   /* Funktionsprototyp */

main() {
     int i;
     for (i=0; i < 10; i++)
         printf( "%d  %20.16f\n", i, wurzel(i) );
}
```

```
double wurzel(double n) {
    double  x = n, alt_x = 0;
    if (n <= 0)
          return 0;

    while ( abs(x - alt_x) > DELTA ) {
          alt_x = x;
          x = (n/x + x)/2;
    }
    return x;
}
```

Aufgabe 2.8: Verbessern Sie die Funktion `wurzel()`*, so dass im Fall eines negativen Radikanden eine entsprechende Fehlermeldung erfolgt.*

Beispiel 2.7:

```
#include <stdio.h>

main() {                              /*  Geheimschrift  */
      int c;
      while ( (c = getchar()) != EOF)
            putchar(c^(-1));
}
```

Dies Programm verändert `c` mit dem Operator `^` (Bitweis-Oder). Dadurch werden Zeichen erzeugt, die u. U. nicht druckbar sind. Daher ist es besser, die Ausgabe in eine Datei zu schreiben mit der Unix-Umleitung

```
./a.out < quelldatei > zieldatei
```

Ein zweiter Aufruf

```
./a.out < zieldatei > zieldatei2
```

"dekodiert" die "kodierte" Datei: `zieldatei2` ist nun identisch mit `quelldatei` – warum?

Aufgabe 2.9: Modifizieren Sie dies Programm, so dass ein beliebiger, jeweils auf eine Abfrage hin eingebbarer Schlüssel verwandt werden kann.

Beispiel 2.8:

```
#include <stdio.h>

main() {                        /* Fuegt Zeilennummern ein */
      int c, d, nl;
      nl = 1;
      printf("%4d %4c", nl, ' ');
      while ((c = getchar()) != EOF)  {
            if (c == '\n') {
                ++nl; putchar(c);
                printf("%4d %4c", nl, ' ');
            }
            else putchar(c);
      }
}
```

Aufgabe 2.10: Ändern Sie das Programm so ab, dass die Zeilennummern als Kommentare in einem C-Programm gedruckt erscheinen, also von / ... */ eingerahmt sind.*