Einführung in die Programmiersprachen C und C++

Markus Kirschmer, Fakultät für Mathematik

Übungsblatt 2

Elementare Datentypen in C und C++

Typ	Beschreibung
`char`	1 Byte, kann „einen Charakter“ (z. B. `'a', '3', '&', ...`) enthalten.
`int`	ganze Zahl, Größe ist implementationsabhängig.
`float`	Dezimalbruch einfacher Genauigkeit.
`double`	Dezimalbruch doppelter Genauigkeit.

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

Typische Wertebereiche (für 64bit-Linux):

Typ	min	max	Bytes
`char`	-128	$+127$	1
`short`	-32768	$+32767$	2
`int`	-2147483648	$+2147483647$	4
`long`	-2^{63}	$+(2^{63}-1)$	8
`float`	$\pm 1.7 \cdot 10^{-38}$	$\pm 3.4 \cdot 10^{+38}$	4
`double`	$\pm 2.2 \cdot 10^{-308}$	$\pm 1.7 \cdot 10^{+308}$	8

Schlüsselwörter von C und C++

sind reserviert und stehen nicht als Bezeichner zur Verfügung.[1]

`and`[1]	`const`	`false`[1]	`not`[1]	`signed`	`typeid`[1]
`and_eq`[1]	`const_cast`[1]	`float`	`not_eq`[1]	`sizeof`	`typename`[1]
`asm`	`continue`	`for`	`operator`[1]	`static`	`union`
`auto`	`default`	`friend`[1]	`or`[1]	`static_cast`[1]	`unsigned`
`bitand`[1]	`delete`[1]	`goto`	`or_eq`[1]	`struct`	`using`[1]
`bitor`[1]	`do`	`if`	`private`[1]	`switch`	`virtual`[1]
`bool`[1]	`double`	`inline`[1]	`protected`[1]	`template`[1]	`void`
`break`	`dynamic_cast`[1]	`int`	`public`[1]	`this`[1]	`volatile`
`case`	`else`	`long`	`register`	`throw`[1]	`wchar_t`[1]
`catch`[1]	`enum`	`mutable`[1]	`reinterpret_cast`[1]	`true`[1]	`while`
`char`	`explicit`[1]	`namespace`[1]	`return`	`try`[1]	`xor`[1]
`class`[1]	`export`[1]	`new`[1]	`short`	`typedef`	`xor_eq`[1]
`compl`[1]	`extern`				

1. Unäre Operatoren:

`+, -` (Vorzeichen), `!` (Negation), `~` (Bit-Komplement), `++, --` (In- und Dekrement, beide als Prä- oder Postoperatoren), `*` (Umleitung, Indirection), `&` (Adressoperator), `sizeof` (Bytegröße eines Datentyps).

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, erst 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: `a` und `b` haben beide den Wert 6.

[1] Diese Schlüsselwörter treten nur in C++ auf.

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-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 $101000 \cong 40$ im Zehnersystem.

`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`.

Operator	Beschreibung	Beispiel
`&`	bitweises 'UND'	`00101000 & 00110010 = 00100000`
`|`	bitweises 'ODER'	`00101000 | 00110010 = 00111010`
`^`	bitweises exklusives 'ODER'	`00101000 ^ 00110010 = 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: `( a > b ) ? a : b;` liefert das Maximum von `a` und `b`.

4. Zuweisungsoperatoren: `=`, Beispiel: `a = 17;`

Eine Zuweisung liefert einen Wert, nämlich den der zugewiesenen Größe, also sind Mehrfachzuweisungen wie `a = b = c = 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).

Liste aller Operatoren in fallender Präzedenz

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

Die unären Operatoren `+, -, *, &` haben höhere Präzedenz als die binären Formen. Die Assoziativität regelt den Fall, daß ein Operand zwischen zwei binären 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`
`a = b += 3` bedeutet `a = (b += 3)`

Beispiel 2.1 Testen Sie z. B. die Wirkungsweise des `?:`-Operators und des `<<`-Operators mit folgendem Programm:

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

int 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 einige der oben angegebene Operatoren auf ihre Wirkungsweise, indem Sie Werte von Ausdrücken ausgeben, in denen die Operatoren auftauchen.*

Arrays 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 Arrays vom Typ char */
/* Zuordnung: Der Zahl 2*i+3 entspricht der Eintrag flags[i]        */
/* Genauer: flags[i] == TRUE  <==>  2*i+3 ist prim                  */

int main() {
```

```
  int i, p = 2, k, size, count = 1;

  printf("Bestimmung aller Primzahlen bis (Zahl>=2) : ");
  scanf("%d", &size);
  size = (size+1)/2-2;

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

  for (i = 0; i <= size; i++) {
    if (flags[i]) {
      /* 2*i+3 ist prim, also streichen wir seine Vielfachen           */
      p = 2*i+3;
      count++;
      for (k=i+p; k<=size; k+=p)
        flags[k] = FALSE;
    }
  }
  printf("\n%d Primzahlen\n",count);
  printf("\nLetzte: %d\n",p);
}
```

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 (=Zeichenfolgen) definiert und verwendet.

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

char *tage[] = { "Sonntag", "Montag", "Dienstag",
          "Mittwoch", "Donnerstag", "Freitag", "Samstag" };
int 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, 4, 2007`*, erfolgt die Ausgabe* `13. April 2007`*.*

Hinweis: Definieren Sie ein Array `char *monat[]` derart, daß z. B. `monat[4]` den Wert `"April"` hat.

Für Ehrgeizige: Erweitern Sie das Programm, so daß die Ausgabe lautet: `Freitag, 13. April 2007`. Ferner soll das Programm alle Monate des eingegebenen Jahres finden, bei denen der 13. ein Freitag ist. Hinweis: `de.wikipedia.org/wiki/Gau%C3%9Fsche_Wochentagsformel`

Zur Syntax und Semantik von printf und scanf:
Vergleiche die Manual-Einträge: `man 3 printf, man scanf`, die Zeilen mit **??** sind Mini-Aufgaben.

Beispiele:
`printf("Backslash gefolgt von n bedeutet \n \"Newline\"")` liefert die Ausgabe

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

`printf("%d", a)` druckt den Inhalt der Integer-Variablen `a` als Dezimalzahl.
`printf("%o", b)` druckt den Inhalt der Integer-Variablen `a` als Oktalzahl.
?? `printf("%x", c)` tut was?
?? `printf("%10.6f", d)` gibt die Fließkommazahl `d` wie aus?
?? Ersetze `f` im letzten Beispiel durch `e, g`. Was geschieht?
`printf("%s", t)` gibt den String `t` aus.
`printf("%c", x)` gibt den Charakter (Zeichen) `x` aus.
?? Was bedeuten die Control-Strings `"%10s", "%-20s", "%-20.4s"` usw.?

Der allgemeine Aufruf von `printf` hat die Form

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

Der Control-String `<string>` wird Zeichen für Zeichen auf den Bildschirm 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 Ausgabeformat fest (in diesem Fall: Feldbreite 7 und 4 Nachkommastellen), 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.

`printf("%c",x)` erwartet die Variable `x` vom Typ `char` und `printf("%s",z)` erwartet in `z` einen String.

Der Rückgabewert von `printf()` ist vom Typ `int` und liefert die Anzahl der geschriebenen Zeichen.

Ähnliche Regeln gelten für die Argumente der Input-Funktion `scanf`.

Beispiel: `scanf("%d", &a)` erwartet (z. B. von der Tastatur) eine ganze Zahl als Eingabe und weist deren Wert der Integer-Variablen `a` zu. Als Argument gibt man nicht `a` selbst an, sondern `&a`, die Adresse von `a`.

Der allgemeine Aufruf von scanf ist von der Form

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

die optionalen Variablen sind die **Adressen** der Variablen `var_i`.

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 Spaces”, d.h. Folgen von Leerzeichen (Blanks), Tabulatorzeichen (Tabs), Zeilenumbrüchen (Newlines) beliebiger Länge (d.h. auch der Länge = 0) im Control-String entspricht White Spaces 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 Variablen (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>
int 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 Zahlen zu untersuchen.

Beispiel 2.3:

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

int power(int,int), fact(int);

int 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 muß vor dem Erstaufruf *deklariert* sein. Dies geschieht entweder durch Angabe ihrer kompletten Definition (d.h. des Quelltexts) oder aber zumindest durch Angabe Ihres *Prototyps* (die Angabe des Funktionsnamens und der Typen ihrer Argumente und des Rückgabewerts), vgl. das folgende Beispiel.

```
/* Quadratwurzel ohne Fehlerausgabe fuer negative Radikanden       */
#include <stdio.h>

#define  DELTA  1.0e-16

double wurzel(double);   /* Funktionsprototyp */

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

double betrag(double x) {
  return x < 0 ? -x : x;
}
```

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

  while ( betrag(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>

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

Dies Programm verändert `c` mit dem Operator `^` (Bitweises-Oder). Dadurch werden evtl. Zeichen erzeugt, die 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>

int 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.*